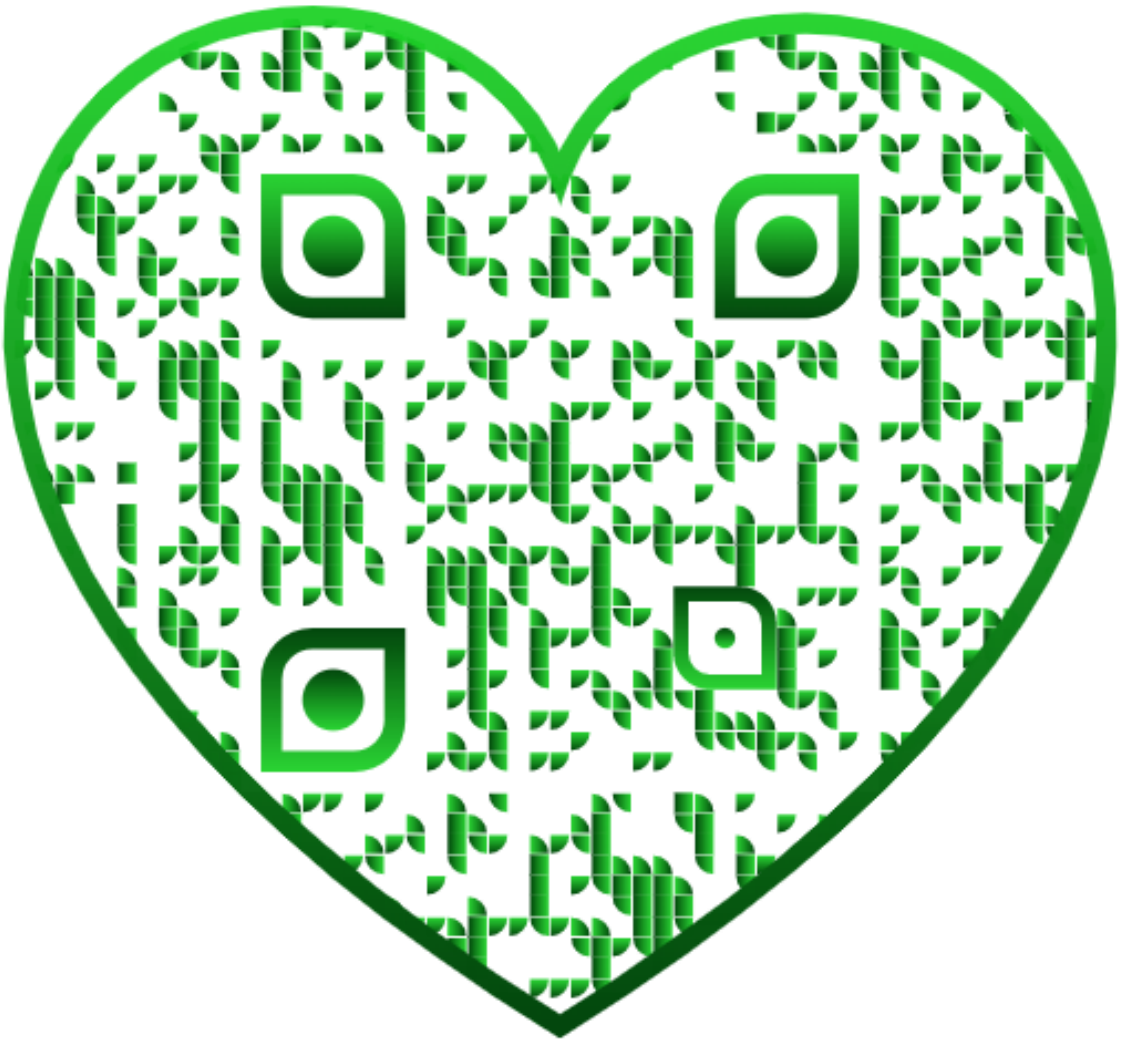
INTELLIGENCE ARTIFICIELLE & INFORMATIQUE THÉORIQUE

2e éd.



J.-M. ALLIOT
T. SCHIEX
P. BRISSET
F. GARCIA



Intelligence artificielle et Informatique théorique

2e édition

Jean-Marc ALLIOT
Ancien élève de Polytechnique
Docteur en Informatique

Thomas SCHIEX
Ancien élève de l'École Centrale des Arts et Manufacture
Docteur en Intelligence artificielle

Pascal BRISSET
Ancien élève de l'École Normale Supérieure de Saint-Cloud
Docteur en Informatique

Frédérick GARCIA
Ancien élève de l'École Nationale Supérieure
de l'Aéronautique et de l'Espace
Docteur en Intelligence artificielle

CÉPADUES-ÉDITIONS
111, rue Nicolas-Vauquelin
31100 TOULOUSE – France
Tél. : 05 61 40 57 36 – Fax : 05 61 41 79 89
(de l'étranger) + 33 5 61 40 57 36 – Fax : + 33 5 61 41 79 89
www.cepadues.com
e.mail: cepadues@cepadues.com

CHEZ LE MÊME ÉDITEUR

I.A. et Informatique théorique ..... *Alliot J.-M., Schiex T.*
Techniques avancées pour le traitement de l'information ..... *Amat J.-L. et Yahiaoui G.*
Acquisition et ingénierie des connaissances ..... *Aussenac-Gilles, Laublet, Raynaud*
Simulation et algorithmes stochastiques :
une introduction avec applications ..... *Bartoli N., Del Moral P.*
L'indicateur de performance : concepts & applications ..... *Berrah L.*
Logique floue. Exercices corrigés
et exemples d'applications ..... *Bouchon-Meunier B., Foulloy L., Ramdani M.*
BDA' 2001 ..... *Collectif*
HCI Aero 2000 ..... *Collectif*
CGIP' 2000 ..... *Collectif*
Le document multimédia – Tome 1 ..... *Collectif*
Le temps, l'espace et l'évolutif – Tome 2 ..... *Collectif*
QCAV 2001 ..... *Collectif*
IHM 2001, 99, 98, 97, 96, 95 ..... *Collectif*
Rencontres francophones sur la logique floue
et ses applications - LFA 2001, 2000, 99, 98, 97, 96, 95 ..... *Collectif*
La CAO par le menu – Lisp par Autocad ..... *Contensou J.-N.*
Induction symbolique numérique à partir de données ..... *Diday E., Kodratoff Y. et al.*
Introduction au modèle de maturité CMM ..... *Dymond K.-M., trad. A. Combelles et al.*
I3 – Revue Information-Interaction-Intelligence ..... *Garbay C., Doucet A., Prade H.*
Traitement parallèle dans les bases de données relationnelles ..... *Hameurlain A. et al.*
Conception de circuits en VHDL ..... *Houzet D.*
Analyse des structures par éléments finis, 3e éd. ..... *Imbert J.-F.*
Guide de la sûreté de fonctionnement ..... *Laprie J.-C.*
Optique et information ..... *Laug M.*
Structures de programmation parallèle ..... *Mauran Ph., Padiou G., Papaix Ph.*
Bases de radiométrie optique 2e éd. ..... *Meyzonette J.-L. , Lépine Th.*
Techniques de synchronisation pour les applications parallèles ..... *Padiou G., Sayah A.*
A la découverte du C++ – 2e éd. ..... *Perrin F.*
Programmation langage C ..... *Rigaud J.-M., Sayah A.*
Votre entreprise sur Internet ..... *Ventre D., David F.*

© CEPAD 2002

ISBN : 2.85428.578.6



Le code de la propriété intellectuelle du 1er juillet 1992 interdit expressément la photocopie à usage collectif sans autorisation des ayants droit. Or, cette pratique en se généralisant provoquerait une baisse brutale des achats de livres, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement serait alors menacée.

Nous rappelons donc que toute reproduction, partielle ou totale, du présent ouvrage est interdite sans autorisation de l éditeur ou du Centre français d exploitation du droit de copie (CFC – 3, rue d Hautefeuille – 75006 Paris).

Dépôt légal : mars 2002

N° éditeur : 578

Il y a ceux qui font quelque chose :
Ils sont trois qui font quelque chose.
Il y a ceux qui ne font rien :
Ils sont dix qui font des conférences
Sur ce que font les trois qui font quelque chose.
Il y a ceux qui croient faire quelque chose
Et ils sont cent qui font des conférences
Sur ce que disent les dix
De ce que font les trois qui font quelque chose.
Et il arrive que l'un des cent dix vienne expliquer
La manière de faire à l'un des trois qui font quelque chose.
Alors l'un des trois intérieurement s'exaspère et
Extérieurement sourit
Mais il se tait car il n'a pas l'habitude de la parole
D'ailleurs, il a quelque chose à faire.

Paul Valéry

Que les trois qui font nous pardonnent...

Remerciements

Il nous aurait été simplement impossible d'écrire cet ouvrage sans la collaboration de nombreuses personnes. Philippe Quéinnec a accompli un travail considérable : il a critiqué en détail le contenu et la présentation de chacun des chapitres. Nous lui sommes redevables bien plus que l'absence de son nom sur la page de couverture ne le laisse paraître. Michel Cayrol et Luis Fariñas Del Cerro ont été les grands « Inspirateurs » de cet ouvrage. Leur esprit habite chaque octet de ce fichier, devenu livre (merci à TEX, LATEX, Donald Knuth et Leslie Lamport).

De nombreux critiques nous ont aidés à améliorer la clarté et la correction de certains chapitres : Léon Bottou (réseaux de neurones), Dominique Colin de Verdière (l'ouvrage entier), Andreas Herzig (logique et résolution), David Lesaint (satisfaction de contraintes), Marc Schoenauer (algorithmes génétiques), Gérard Verfaillie (satisfaction de contraintes), Jean-Christophe Weill (les jeux et leur programmation). Les erreurs qui peuvent subsister sont nôtres. Nous remercions également Stéphane Chatty, Nicolas Courtel, Marie-Christine Lagasquie et Olivier Palmade pour toute l'aide qu'ils ont pu nous apporter, dans bien des domaines, Hervé Gruber pour son travail sur les algorithmes génétiques.

De nombreuses autres personnes ont participé, par leur aide, leurs encouragements et leurs remarques à la réalisation de cet ouvrage : Daniel Azéma, Philippe Balbiani, Jean-François Bosc, Bernard Brémond, Béatrice Cazard, Jean Chassaing, Olivier Dessoude, Sylvie et Damien Figarol, Eric Fruit, Jean-Marc Garot, Georges Joly, Tibor Kökeny, Alexandre Lefebvre, David Martinez, Laurent Miclet, Paul-Henri Mourlon, François Rouaix, Michel Sabatier, Frédéric Vardon, Guillaume Watier...

Certains sujets difficiles n'auraient pas pu être traités sans la collaboration de quelques « spécialistes », détenteurs généreux d'informations rares et vitales qui ont contribué à l'originalité de certains chapitres : le professeur William Armstrong a accepté avec une grande gentillesse de faire une conférence sur les réseaux de neurones logiques à l'École Nationale de l'Aviation Civile, Gregory Chaitin nous a communiqué l'ensemble de ses écrits et a discuté nombre de nos idées, Feng-Hsiung Hsu a répondu avec une grande disponibilité à toutes nos questions concernant l'architecture et les concepts qui ont permis le développement de DEEP THOUGHT, *William Rapaport nous a aimablement communiqué d'utiles références concernant les domaines recouverts par les sciences cognitives, le professeur Jonathan Schaeffer nous a très aimablement fourni l'ensemble des articles concernant* CHINOOK, *et les discussions sur l'IA et les ordinateurs que nous avons pu avoir ont contribué largement à asseoir nos opinions.*

Nous remercions enfin les organismes qui nous ont hébergés durant la période de rédaction de ce document : le centre d'études de la navigation aérienne (CENA), l'école nationale de l'aviation civile, le centre d'études et de recherches de Toulouse, établissement de l'ONERA (office national d'études et de recherches aérospatiales) et l'institut national de la recherche agronomique (INRA).

Table des matières

Avant-propos 21

1 Définir l'Intelligence Artificielle 25
- 1.1 Qu'est-ce que l'intelligence artificielle ? . . . 25
- 1.2 Historique . . . 26
  - 1.2.1 La préhistoire : 1945-1955 . . . 26
  - 1.2.2 Les débuts : 1955-1970 . . . 26
  - 1.2.3 La spécialisation : 1970-1980 . . . 28
  - 1.2.4 Une reconnaissance : 1980-1990 . . . 28
- 1.3 Les différentes approches de l'IA . . . 30
  - 1.3.1 L'approche cognitive . . . 30
  - 1.3.2 L'approche pragmatiste . . . 31
  - 1.3.3 L'approche connexionniste . . . 32
  - 1.3.4 Convergence de ces approches ? . . . 33
- 1.4 Notre définition de l'IA . . . 34
- 1.5 Problèmes typiques d'IA . . . 35
  - 1.5.1 Les problèmes spécifiables et difficiles . . . 36
  - 1.5.2 Problèmes recouvrant un domaine précis . . . 37
  - 1.5.3 Autres domaines liés à l'IA . . . 38

I Logique mathématique, résolution 39

2 Le calcul propositionnel 41
- 2.1 Méthodologie . . . 41
- 2.2 Introduction informelle . . . 42
- 2.3 Définitions . . . 42
- 2.4 Théorie des modèles . . . 43
  - 2.4.1 Introduction . . . 43
  - 2.4.2 Définition formelle en calcul propositionnel . . . 43
  - 2.4.3 Application pratique : les tables de vérité . . . 44
  - 2.4.4 Tautologies . . . 45
  - 2.4.5 Substitution . . . 46
  - 2.4.6 La notion de conséquence valide . . . 47
- 2.5 Théorie de la démonstration . . . 48
  - 2.5.1 Axiomatique du calcul propositionnel . . . 48
  - 2.5.2 Démonstrations . . . 49

2.5.3 Déduction 50
2.6 Propriétés fondamentales 50
2.6.1 Adéquation 51
2.6.2 Consistance 51
2.6.3 Complétudes 51
2.6.4 Décidabilité 52
2.6.5 Remarques 52

3 Le calcul des prédicats 55
3.1 Introduction 55
3.2 Définitions 56
3.2.1 Variables liées, variables libres 56
3.2.2 Clôture d'une formule 57
3.3 Théorie de la démonstration 58
3.3.1 Axiomatique du calcul des prédicats 58
3.3.2 Démonstration – Déduction 59
3.3.3 Théorème de la déduction 59
3.4 Théorie des modèles 59
3.4.1 Introduction informelle 59
3.4.2 Interprétation 60
3.4.3 Formule valide 61
3.4.4 Exemple de table de vérité 61
3.4.5 Satisfiabilité, insatisfiabilité et conséquence 61
3.4.6 Le problème des domaines infinis 62
3.5 Propriétés fondamentales 63
3.5.1 Adéquation - Consistance 63
3.5.2 Complétude 63
3.5.3 Décidabilité 63
3.5.4 Remarques 63

4 Les machines de Turing 65
4.1 Introduction 65
4.2 Machine de Turing déterministe 66
4.3 Notion de calcul 68
4.3.1 Calcul d'un argument et fonction calculable 68
4.3.2 Exemple : $f(x) = x + 1$ 69
4.4 Réalisation pratique 70
4.5 Les machines de Turing non déterministes 70
4.6 Remarques 71

5 Les systèmes formels 73
5.1 Un peu d'histoire 73
5.2 Les problèmes des mathématiques 75
5.2.1 Les malheurs de la géométrie euclidienne 75
5.2.2 L'infini 76
5.2.3 Les ensembles 76
5.2.4 Les paradoxes 77
5.3 Intuitionnistes contre formalistes 79

5.3.1 Les intuitionnistes . . . 79
5.3.2 Les formalistes . . . 80
5.4 L'arithmétique formelle . . . 80
5.5 Décidabilité, complétude et consistance . . . 83
5.5.1 Décidabilité et calculabilité . . . 84
5.5.2 Construction d'une fonction non-calculable . . . 84
5.5.3 Indécidabilité de $\mathbb{N}$ . . . 87
5.5.4 Incomplétude de $\mathbb{N}$ . . . 87
5.5.5 Non démontrabilité de la consistance de $\mathbb{N}$ . . . 89
5.6 Les conséquences . . . 90
5.7 Faut-il jeter la logique ? . . . 91

6 Calcul propositionnel et résolution 93
6.1 Introduction . . . 93
6.2 Principe de déduction . . . 93
6.2.1 Résultats préliminaires . . . 93
6.2.2 Interprétation du résultat . . . 94
6.2.3 Les arbres sémantiques . . . 94
6.2.4 Algorithme de Quine . . . 94
6.2.5 Algorithme de réduction . . . 95
6.3 Formes normales . . . 95
6.3.1 Théorème de normalisation . . . 95
6.3.2 Exemple . . . 96
6.3.3 Vocabulaire . . . 97
6.3.4 Algorithme de Davis et Putnam . . . 97
6.3.5 L'évaluation sémantique . . . 98
6.3.6 Révision rapide . . . 101
6.4 Principe de résolution . . . 102
6.4.1 Théorème de résolution . . . 102
6.4.2 Application . . . 102
6.5 Les clauses de Horn . . . 103
6.5.1 Résultats généraux . . . 103
6.5.2 Clauses de Horn et résolution . . . 103
6.5.3 Exemple de résolution . . . 104
6.5.4 Interprétation intuitive . . . 104

7 Calcul des prédicats et résolution 105
7.1 Introduction . . . 105
7.2 Formes normales . . . 105
7.2.1 Formes prénexes . . . 105
7.2.2 Exemple . . . 106
7.3 Skolémisation . . . 106
7.3.1 Formes de Skolem . . . 106
7.3.2 Algorithme de Skolémisation . . . 107
7.3.3 Exemple . . . 107
7.3.4 Démonstration . . . 107
7.3.5 Forme standard . . . 108
7.4 Théorème de Herbrand . . . 108

7.5 Unification . . . 110
7.5.1 Substitution . . . 110
7.5.2 Unification . . . 111
7.6 Principe de résolution . . . 112
7.6.1 Algorithme et exemple d'application . . . 112

8 Les logiques non-classiques 115
8.1 Logiques faibles . . . 115
8.1.1 La logique absolue A . . . 115
8.1.2 La logique positive P . . . 116
8.1.3 La logique minimale M . . . 117
8.1.4 La logique intuitionniste J . . . 117
8.2 Introduction aux logiques modales . . . 117
8.3 Les modalités . . . 118
8.4 Logique aléthique . . . 119
8.4.1 Le système T . . . 119
8.4.2 Le système S4 . . . 120
8.4.3 Le système S5 . . . 120
8.5 Théorie des modèles . . . 120
8.5.1 Les tables de vérité . . . 121
8.5.2 Sémantique des mondes possibles . . . 121
8.5.3 Interprétation intuitive . . . 121
8.5.4 Un modèle de S4 . . . 122
8.5.5 Un modèle de S5 . . . 122
8.6 Adéquation, consistance, complétude . . . 122
8.7 Une logique modale du premier ordre . . . 123
8.8 Interprétation des logiques modales . . . 123
8.8.1 Une logique de la connaissance . . . 123
8.8.2 Une logique de la croyance . . . 124
8.9 Extensions . . . 124
8.10 Les logiques temporelles . . . 124
8.10.1 Approche intuitive . . . 125
8.10.2 Premiers éléments . . . 125
8.10.3 L'ultériorité . . . 126
8.10.4 Ordre partiel des dates . . . 126
8.10.5 Ordre total des dates . . . 127

II Éléments d'informatique théorique 129

9 Théorie des langages formels 131
9.1 Introduction . . . 131
9.2 Définitions générales . . . 131
9.3 Systèmes de réécriture . . . 132
9.4 Grammaires . . . 133
9.4.1 Éléments de base . . . 133
9.4.2 Exemples . . . 134
9.4.3 Formes normales . . . 135

9.5 Les systèmes de Post . . . 135
9.5.1 Une axiomatique du calcul propositionnel . . . 136
9.5.2 Propriétés . . . 137
9.6 Algorithme de Markov . . . 137

10 La calculabilité 139
10.1 Introduction . . . 139
10.2 Problème de reconnaissance . . . 139
10.3 Le problème de correspondance de Post . . . 140
10.3.1 Énoncé intuitif du problème . . . 140
10.3.2 Démonstration de l'indécidabilité . . . 140
10.3.3 Autres résultats sur le PCP . . . 142
10.3.4 Autres résultats liés au PCP . . . 143
10.4 Les solutions d'équations diophantiennes . . . 144
10.4.1 Définitions . . . 144
10.4.2 Résultats fondamentaux . . . 144
10.4.3 Le dixième problème de Hilbert . . . 146
10.4.4 Conséquences . . . 146
10.5 Caractère aléatoire d'un programme . . . 147
10.5.1 Hasard et calculabilité . . . 147
10.5.2 Probabilité d'arrêt d'un programme . . . 149
10.5.3 Complexité organisée . . . 149

11 La complexité 151
11.1 Introduction . . . 151
11.2 Définitions fondamentales . . . 151
11.2.1 Problèmes de décision et langages . . . 151
11.3 Problèmes faciles et intraitables . . . 154
11.3.1 Problèmes polynomiaux . . . 154
11.3.2 Les problèmes prouvés intraitables . . . 155
11.3.3 Conclusion . . . 155
11.4 Résultats généraux sur les problèmes NP . . . 156
11.4.1 Problèmes NP . . . 156
11.4.2 Problèmes NP-complets . . . 156
11.4.3 Le théorème de Cook . . . 158
11.4.4 Exemples de problèmes NP-complets . . . 163
11.4.5 Preuve de NP-complétude . . . 165
11.5 Schémas d'approximation . . . 166
11.5.1 Principes généraux . . . 167
11.5.2 Le problème du voyageur de commerce . . . 167
11.5.3 Problèmes admettant des ε-approximations . . . 168
11.6 Problèmes pseudo-polynomiaux . . . 169
11.7 La cryptographie : une application . . . 170
11.8 Complexité en terme d'espace . . . 171
11.9 Autres classes . . . 172

12 λ-calcul 175
12.1 Introduction . . . 175
12.2 Définitions . . . 176
12.2.1 Variables libres ou liées . . . 177
12.3 Calcul sur les termes du λ-calcul . . . 178
12.3.1 Substitution . . . 178
12.3.2 α-équivalence . . . 178
12.3.3 β-réduction . . . 179
12.3.4 Formes normales . . . 180
12.3.5 Théorème de Church-Rosser . . . 181
12.3.6 Ordre de réduction . . . 181
12.3.7 β-équivalence . . . 183
12.3.8 Extensionnalité et η-réduction . . . 183
12.4 Expression de fonctions récursives . . . 184
12.4.1 Les combinateurs de point fixe . . . 185
12.5 Utilisation du λ-calcul . . . 187
12.5.1 Les booléens et opérations booléennes . . . 187
12.5.2 Les doublets . . . 188
12.5.3 Les nombres entiers . . . 189
12.6 Termes du λ-calcul typé . . . 192
12.6.1 λ-calcul simplement typé . . . 192
12.6.2 Affectation de type en λ-calcul non typé . . . 195
12.6.3 Un système de typage polymorphique . . . 199

III Techniques de l'Intelligence Artificielle 203

13 Méthodes faibles 205
13.1 Introduction . . . 205
13.2 Espaces d'états . . . 205
13.2.1 Principes généraux . . . 205
13.2.2 Exemples . . . 206
13.3 Systèmes de production . . . 208
13.3.1 Raisonnement en chaînage avant . . . 208
13.3.2 Raisonnement en chaînage arrière . . . 208
13.3.3 Monotonie et commutativité . . . 209
13.3.4 Exemples . . . 210
13.4 Représentation formelle . . . 211
13.4.1 Arbres . . . 212
13.4.2 Graphes . . . 212
13.4.3 Graphes ET-OU . . . 213
13.5 Stratégies de résolution . . . 213
13.5.1 Algorithme du British Museum . . . 213
13.5.2 Recherche en profondeur et retour-arrière . . . 214
13.5.3 Recherche en largeur . . . 215
13.5.4 La notion d'heuristique . . . 215
13.5.5 L'escalade . . . 216
13.5.6 Recherche *meilleur en premier* : graphe OU . . . 218

13.5.7 Recherche *meilleur en premier* : graphe ET-OU 221
13.6 Analyse d'un problème 221
13.6.1 Calculabilité et complexité du problème 222
13.6.2 Problèmes décomposables 222
13.6.3 Problèmes prévisibles 223
13.6.4 Problèmes ignorables ou récupérables 223
13.6.5 Optimisation ou satisfiabilité 224
13.7 Conclusion 225

14 Problèmes de satisfaction de contraintes 227
14.1 Introduction 227
14.2 Langage et notations 228
14.3 Sémantique 231
14.4 Énoncés de problèmes 235
14.4.1 Algorithme standard pour la satisfaction 235
14.5 Consistances locales et filtrage 237
14.5.1 Filtrages 238
14.5.2 Arc-consistance 238
14.5.3 Chemin-consistance 243
14.5.4 Autres propriétés de consistances locales 243
14.6 Sophistiquer l'algorithme Backtrack 244
14.6.1 Ordres et heuristiques 245
14.6.2 Backtrack intelligent 249
14.6.3 Mémorisation de contraintes 250
14.6.4 Méthodes prospectives 253
14.6.5 Remise en cause de l'ordre d'exploration 254
14.6.6 Méthodes hybrides 256
14.7 Classes polynomiales et décomposition 257
14.7.1 Classes polynomiales structurelles 257
14.7.2 Classes polynomiales micro-structurelles 260
14.8 Extensions du cadre classique 260
14.8.1 Contraintes globales 261
14.8.2 Satisfaction partielle 262
14.8.3 CSP dynamiques 262
14.9 Quelques problèmes 263

15 La programmation des jeux 267
15.1 Introduction 267
15.2 Principe minimax (négamax) 268
15.2.1 Fonctions d'évaluation 269
15.2.2 Algorithme minimax 269
15.2.3 Algorithme α-β 270
15.3 L'algorithme SSS* 272
15.4 L'algorithme Scout 277
15.5 La pratique après la théorie 278
15.5.1 Contrôle du temps et α-β 278
15.5.2 Tables de transposition 279
15.5.3 α-β avec mémoire 281

15.5.4 Fenêtre réduite et MTD(f) . . . 283
15.5.5 Paralléliser un algorithme α-β . . . 284
15.5.6 Pour conclure ? . . . 286
15.6 Othello . . . 286
15.6.1 Les ouvertures . . . 286
15.6.2 Les finales . . . 287
15.6.3 Le milieu de partie . . . 287
15.6.4 Othello et apprentissage . . . 288
15.7 Les échecs . . . 289
15.7.1 Les études des psychologues . . . 290
15.7.2 Les ouvertures . . . 294
15.7.3 Les finales . . . 295
15.7.4 Le milieu de partie . . . 299
15.8 Le bridge . . . 304
15.8.1 Les annonces . . . 304
15.8.2 Le jeu de la carte . . . 305
15.9 Autres jeux . . . 307
15.10 Les jeux vus par Conway . . . 310
15.10.1 Quels jeux ? . . . 310
15.10.2 Quelques jeux de Conway . . . 312
15.10.3 Les nombres . . . 315
15.10.4 Les jeux impartiaux et les nimbres . . . 315
15.10.5 Conclusion . . . 318

16 Les systèmes experts 321
16.1 Introduction . . . 321
16.2 La représentation des connaissances . . . 322
16.2.1 Logique des prédicats . . . 322
16.2.2 Logique modale . . . 323
16.2.3 Logique temporelle . . . 325
16.2.4 Logique multivaluée et logique floue . . . 326
16.2.5 Logique des défauts . . . 327
16.2.6 Réseaux sémantiques . . . 329
16.2.7 Dépendance conceptuelle . . . 331
16.2.8 Frames . . . 332
16.2.9 Scripts . . . 332
16.3 SE à base de règles . . . 333
16.3.1 Les moteurs d'inférence . . . 334
16.3.2 Systèmes de maintien de cohérence . . . 335
16.3.3 Langages d'implantation . . . 337
16.4 Exemples de réalisation . . . 337
16.5 Critique des systèmes experts . . . 339
16.6 Les systèmes experts par modèles . . . 340
16.7 Mythes et légendes . . . 341
16.7.1 Les mythes . . . 341
16.7.2 Les légendes . . . 342
16.8 Conclusion . . . 342

IV Les langages de l'IA 345

17 Programmation fonctionnelle : ML 347
17.1 Introduction . . . 347
17.2 Le langage OCaml . . . 347
17.2.1 Un premier survol . . . 348
17.2.2 Syntaxe et typage . . . 349
17.2.3 Évaluation . . . 350
17.2.4 Nommage . . . 351
17.2.5 Ordre supérieur . . . 352
17.2.6 Types de base et conditionnelle . . . 352
17.2.7 Pattern-matching . . . 352
17.2.8 Fonctions récursives . . . 353
17.2.9 Listes . . . 354
17.2.10 Itérateurs . . . 354
17.2.11 Définition de types . . . 355
17.2.12 Le compilateur . . . 356
17.3 Exemples pour l'IA . . . 357
17.3.1 Évaluation d'une formule propositionnelle . . . 357
17.3.2 Codage d'un algorithme A^* . . . 358
17.4 Aller plus loin avec OCaml . . . 360
17.4.1 Modularité . . . 360
17.4.2 Style impératif . . . 360
17.4.3 Programmation orientée objet . . . 361

18 Programmation logique : PROLOG 363
18.1 Présentation . . . 363
18.2 Terminologie PROLOG . . . 364
18.3 La syntaxe . . . 365
18.3.1 Les notions de OU et de ET . . . 366
18.3.2 Premier exemple d'exécution . . . 366
18.4 Moteur d'inférence . . . 367
18.4.1 Fonctionnement simplifié . . . 367
18.4.2 Le retour-arrière et la notion de points de choix . . . 368
18.4.3 La résolution : un parcours d'arbre ET-OU . . . 370
18.4.4 L'unification . . . 372
18.5 Les méta-opérations . . . 374
18.5.1 Modifier le contrôle : le *cut* . . . 374
18.5.2 Les prédicats prédéfinis . . . 376
18.5.3 L'évaluation gelée . . . 376
18.5.4 Le problème de la négation . . . 377
18.6 PROLOG et parallélisme . . . 377
18.7 Programmation logique avec contraintes . . . 378
18.7.1 Le schéma CLP($\mathcal{X}$) . . . 379
18.7.2 Le langage PROLOG III . . . 382
18.7.3 Conclusion . . . 384

V Apprentissage 387

19 Apprentissage Symbolique Automatique 389
19.1 Présentation . . . 389
19.2 Différentes formes d'apprentissage . . . 390
19.3 EBL . . . 391
19.4 SBL . . . 392
19.4.1 Les éléments de base . . . 392
19.4.2 La généralisation . . . 392
19.5 Conclusion . . . 393

20 Les réseaux de neurones 395
20.1 L'argument physiologique . . . 395
20.2 Les mémoires associatives et le modèle de Hopfield . . . 396
20.2.1 Modèle mathématique . . . 396
20.2.2 Mémorisation d'un vecteur binaire . . . 396
20.2.3 Mémorisation de plusieurs vecteurs binaires . . . 397
20.2.4 Capacité de stockage . . . 398
20.3 Les réseaux à sens unique . . . 400
20.3.1 Les réseaux à deux couches . . . 400
20.3.2 Les réseaux multi-couches . . . 402
20.4 Astuces techniques . . . 404
20.4.1 La fonction d'activation . . . 404
20.4.2 Le taux d'apprentissage . . . 405
20.4.3 Comment briser la symétrie . . . 405
20.4.4 Comment éviter les minima locaux . . . 406
20.5 Les réseaux de neurones logiques (ALN) . . . 406
20.5.1 Présentation générale . . . 406
20.5.2 Apprentissage . . . 406
20.5.3 Apprentissage de fonctions numériques . . . 408
20.5.4 Avantages des ALN . . . 409
20.6 Les réseaux de Kohonen . . . 410
20.6.1 Principes généraux . . . 410
20.6.2 Calcul de la valeur de sortie . . . 410
20.6.3 Apprentissage . . . 412
20.7 Conclusion . . . 412

21 Algorithmes évolutionnaires 415
21.1 Introduction . . . 415
21.2 Algorithmes génétiques . . . 416
21.2.1 Principe général . . . 416
21.2.2 Un exemple . . . 416
21.2.3 Discussions des divers paramètres . . . 418
21.3 Convergence théorique . . . 418
21.3.1 Généralités . . . 418
21.3.2 Théorie des schémas . . . 419
21.4 Exemple complet d'application . . . 422
21.4.1 Principe . . . 422

21.4.2 Modélisation du problème . . . 422
21.4.3 Application de l'algorithme génétique . . . 423
21.4.4 Conflit à 5 avions . . . 424
21.5 Techniques avancées . . . 425
21.5.1 Utilisation d'un codage et d'opérateurs spécialisés . . . 425
21.5.2 Opérateurs de réorganisation . . . 427
21.5.3 Le scaling ou mise à l'échelle . . . 428
21.5.4 Modes de sélection, tournoi . . . 429
21.5.5 Le sharing . . . 429
21.5.6 Fonctions partiellement séparables : croisement adapté . . . 430
21.5.7 Structures de données complexes . . . 432
21.5.8 Recherche multi-objectifs . . . 435
21.5.9 Algorithmes génétiques et coévolution . . . 436
21.6 Stratégies d'évolution . . . 438
21.7 Programmation évolutionnaire . . . 438
21.8 La programmation génétique . . . 439
21.9 Conclusion . . . 441

22 Apprentissage par renforcement 443
22.1 Introduction . . . 443
22.2 Les processus décisionnels de Markov . . . 445
22.2.1 Définition d'un processus décisionnel de Markov . . . 445
22.2.2 Les stratégies et les critères de performance . . . 447
22.2.3 Optimalité et fonctions de valeur . . . 448
22.2.4 Algorithmes de résolution des PDM . . . 450
22.2.5 Observabilité du processus . . . 452
22.3 Les méthodes de l'apprentissage par renforcement . . . 453
22.3.1 Méthodes directes et indirectes . . . 454
22.3.2 L'algorithme Q-learning . . . 455
22.3.3 Le dilemme exploration/exploitation . . . 456
22.3.4 La méthode des différence temporelles : TD(λ) . . . 458
22.4 Apprentissage par renforcement et généralisation . . . 464
22.4.1 Représentation paramétrée de la fonction de valeur . . . 465
22.4.2 Les représentations différentiables . . . 466
22.4.3 L'apprentissage de structures d'approximation . . . 471
22.4.4 Méthodes de gradient dans l'espace des stratégies . . . 473
22.5 Conclusion . . . 473

VI Conclusion 477

23 Conclusion 479
23.1 Les critiques . . . 480
23.1.1 Les erreurs de jugement . . . 480
23.1.2 Concepts et modèles : le réductionnisme . . . 481
23.1.3 L'erreur de perspective . . . 483
23.1.4 L'influence des résultats métamathématiques . . . 484
23.1.5 L'argument quantique . . . 484

23.1.6 Critique théologique 484
23.1.7 Critique psychologique 485
23.1.8 Critique biologique 489
23.1.9 Optimalité de l'évolution 490
23.2 Une IA est-elle impossible ? 491
23.3 Qu'en penser ? 492
23.3.1 Différentes attitudes vis-à-vis de l'IA 492
23.3.2 Un avenir pour l'IA 493
23.4 L'IA : une grande illusion ? 494
23.4.1 Rapide analyse historique 494
23.4.2 L'IA, en dernier recours 498
23.5 Addendum à l'édition 2002 499

Bibliographie essentielle 501

Bibliographie 505

Index 533

Préface

S'attaquer aujourd'hui simultanément à l'Intelligence Artificielle et à l'Informatique Théorique est une œuvre titanesque. Bien que nous ne partagions pas tous les points de vue des auteurs (parfois un peu provocateurs), nous pensons qu'ils ont accompli un travail considérable. Nous connaissons assez bien les deux coupables (Thomas Schiex a fait sa thèse sous la responsabilité de Michel Cayrol et Jean-Marc Alliot sous celle de Luis Fariñas del Cerro), et savons que pour eux la découverte de l'informatique a été une révélation, quant à celle de l'Intelligence Artificielle, ils ne sont pas encore au bout de leurs étonnements !

Il s'agit donc d'un ouvrage écrit par deux passionnés qui ont beaucoup appris et compris en le rédigeant. Leur propos est extrêmement ambitieux car le terrain couvert est immense. C'est, à notre connaissance, la première fois, qu'en France, un ouvrage aborde des sujets aussi divers que l'algorithme SSS*, la théorie des jeux de Conway ou les algorithmes génétiques...

Nous pensons qu'ils ont su faire sourdre leur passion entre les lignes et que leur objectif secret est de vous inoculer le virus qui se trouve dans ces pages et qui infecte Jean-Marc et Thomas depuis bien longtemps. Nous espérons que ce livre aura beaucoup de succès, qu'il contribuera à faire mieux connaître quelques joyaux comme la théorie des jeux façon BCG (Berlekamp, Conway & Guy) et qu'il vous donnera l'envie d'aller encore plus loin. Pour conclure, nous souhaiterions appliquer à cet ouvrage la préface de *Introduction à la théorie des langages de programmation* de B. Meyer (InterÉditions) traduit par J. F. Picardat, préface de M. Bertrand Meyer qui s'applique parfaitement au livre de Jean-Marc Alliot et de Thomas Schiex :

« Peut-être même fournira-t-il quelques munitions à ceux qui tentent aujourd'hui de dessiller les yeux des responsables des programmes scolaires et universitaires français. La conception que l'on se fait en haut lieu du rôle de l'informatique est en effet, semble-t-il, bien étrange : dans les lycées, enseignons l'emploi des tableurs ; dans les classes préparatoires, disent les instructions, jetez une pincée d'algorithmique, mais gardez-vous bien d'introduire la récursion ! Sujet fort subversif certes, et dont on ne saurait trop souligner le danger : qu'adviendrait-il de nos valeurs morales et de l'ordre établi si l'on permettait à des turlupins de dix-huit ans d'apercevoir l'infini ? ... »

Souhaitons que cet infini soit le vôtre grâce à ces deux « turlupins » que sont Jean-Marc et Thomas. Merci à eux.

Michel Cayrol et Luis Fariñas Del Cerro

Avant-propos

L'informatique est certainement le domaine qui, dans les quarante dernières années, a connu la progression la plus spectaculaire. D'une part, le matériel a évolué à une incroyable vitesse : en quatre décennies des machines énormes effectuant une centaine d'opérations par seconde, disposant de capacité de stockage de quelques milliers de caractères, valant plusieurs millions d'euros sont devenues des micro-ordinateurs exécutant des centaines de millions d'opérations par seconde, équipés de mémoire de plusieurs centaines de millions de caractères et valant quelques milliers d'euros.

Parallèlement, l'informatique a atteint le statut d'une science à part entière. On a ainsi vu apparaître, ou se développer dans de nouvelles directions, la théorie de la calculabilité, la théorie de la complexité, les théories de programmation logique et fonctionnelle, les théories des graphes et des jeux, des théories rendant compte du parallélisme, de la programmation objet, ou des mécanismes de distribution de processus, en un mot ce qu'il convient d'appeler aujourd'hui une science informatique. Ce développement foudroyant a eu des retombées directes sur les techniques de programmation et de conception, mais aussi sur la façon d'aborder les problèmes que l'on souhaite « informatiser ».

La rapidité de cette évolution a pris de cours aussi bien le grand public, que les professionnels ou les enseignants :

- Le grand public a été et reste la victime de ce que nous appellerons « l'idéologie des machines intelligentes et omniscientes ». On trouve un écho de cette idéologie dans le terme *intelligence artificielle*, terme bien impropre comme nous nous emploierons à le montrer : croire en la possibilité de machines intelligentes ou raisonnant « comme un être humain » relève de l'acte de foi et ne repose sur aucun fondement scientifique. La thèse contraire n'est d'ailleurs pas plus démontrable ; tout au plus peut-on penser que si jamais on parvient à réaliser une forme d'intelligence artificielle, elle s'appuiera sur un matériel et sur des techniques fondamentalement différentes de celles que nous connaissons aujourd'hui : les différences profondes de nature, de structure et de fonctionnement de l'esprit humain et d'une machine informatique classique expliquent l'échec répété des prophéties annonçant la venue d'ordinateurs intelligents. Il ne faut certes pas rejeter les résultats obtenus : les partisans de cette approche ont été les alchimistes d'une nouvelle science : en cherchant la pierre philosophale qu'est l'intelligence artificielle, ils ont contribué à poser et développer les bases de la nouvelle science informatique, mais il faut aujourd'hui s'employer à consolider ces résultats et à leur donner une présentation rigoureuse.
- Les professionnels se sont, eux-aussi, laissés bien souvent dépasser par la rapidité de l'évolution de l'informatique. Les « décideurs », dans nombre d'entreprises, restent

circonspects ou franchement incrédules face à la théorisation de l'informatique moderne. D'autres au contraire se laissent abuser par des phénomènes de mode ou de grands discours creux. D'autre part, les programmeurs, les responsables du développement des programmes informatiques de nombre de sociétés, restent bloqués dans des schémas vieux de vingt ans, se révèlent incapables d'évoluer et sont même hostiles à toute idée de changement.

– La formation des informaticiens faite il y a vingt ans était bien souvent très informelle. L'informatique a été enseignée comme une technique tenant du bricolage et de la débrouillardise. Une formation technique présente de nombreux avantages immédiats[1] : l'individu est plus rapidement à même d'être « utilisable » et « rentable » pour un éventuel employeur. Mais dans un domaine évoluant aussi vite que l'informatique, il s'agit d'un calcul à court terme : à la première évolution importante, le technicien sera privé de toute base théorique à laquelle se rattacher : il se retranchera alors dans une attitude visant à défendre « son » système d'exploitation, « son » langage, « sa » méthode de conception, « sa » méthode de programmation, « son » logiciel... On pourrait penser qu'instruit par ce genre de situation, un effort aurait été fait pour la formation d'informaticiens. Or il n'en est rien. À l'exception des universités qui ont rapidement créé des filiales licence-maîtrise-DEA dotées de programmes couvrant l'essentiel, l'enseignement de l'informatique en France peut être considéré encore aujourd'hui comme sinistré, du primaire aux écoles d'ingénieurs. Des plans informatiques absurdes comme l'opération 10000 micros ne pouvaient que manquer leur objectif : savoir utiliser le clavier d'un ordinateur relève de la bureautique, savoir le programmer de l'informatique. Donner un accès à des machines facilitera peut-être l'agilité manuelle, mais certainement pas les capacités de conceptualisation et de théorisation nécessaires à tout enseignement scientifique. Le véritable enseignement de l'informatique dans le primaire et le secondaire passait par la formation de maîtres et de professeurs spécialisés. Les méthodes actuellement en place n'aboutissent pas à une formation, mais plutôt à une déformation des élèves qui la subissent. D'autre part, la formation informatique des élèves et futurs élèves des écoles d'ingénieurs est bien souvent incomplète et peu formelle. Pourquoi ?

    1. Un des problèmes vient de la place de la logique dans la science en France. Comme le fait remarquer Pascal Engel (Engel 1989), la logique est presque absente du débat philosophique et mathématique français, alors qu'elle oc-

---

1 Il serait bon de s'interroger sur les avantages comparés de la primauté donnée à la science fondamentale ou à la technique. L'historien allemand Rudolf Von Thadden déclarait récemment à propos de la France (Thadden 1991) : « Vous êtes prisonniers de votre système scolaire. Il est axé sur les premiers de classe, le nôtre sur l'idée de produire des masses de gens capables, même dans des postes subalternes. L'Allemagne gagne avec des seconds. » Cette attitude est très proche de celle des Japonais dont la recherche fondamentale est peu développée, mais dont la recherche appliquée et la technologie sont au sommet. Il faudrait donc s'interroger sur la notion de propriété intellectuelle et de propriété technique. Il est possible de protéger des techniques par des brevets, mais toute découverte scientifique fondamentale est publiée et disponible gratuitement, même si aux USA certains algorithmes (donc activité intellectuelle) sont protégés par des brevets (un exemple classique est le système de cryptage RSA). Nous sommes bien conscients de la fragilité des positions techniques face aux pressions économiques et sociales. Une solution, techniquement "meilleure" que les autres, n'a de chance de survivre que si elle parvient d'une part à trouver une justification économique évidente, et si elle peut d'autre part échapper aux pressions corporatistes qui peuvent s'exercer contre elle. Ce n'est pas pour autant qu'il faut renoncer à chercher la solution optimale. Il faut simplement se rendre compte que le rôle du chercheur ou du technicien n'est plus qu'un rôle de conseil. Il n'a plus, aujourd'hui, aucun pouvoir de décision face aux agents politiques, économiques et sociaux.

cupe le devant de la scène dans ces deux domaines dans les pays anglo-saxons. On ne peut que constater, et déplorer hélas, cet état de fait, lié à des raisons historiques, aussi bien dans le cas des mathématiques[2] que de la philosophie. De Condillac à Dieudonné en passant par Poincaré, l'école française a considéré la logique comme un domaine soit extérieur aux mathématiques, soit trivial et ne demandant pas d'études particulières. Cela est d'autant plus regrettable qu'une bonne part de l'intelligence artificielle puise aujourd'hui ses sources dans la logique, et que la réapparition de l'enseignement de la logique n'a pu se faire que dans les facultés à travers les DEA d'informatique. Dans les écoles d'ingénieurs ne préparant pas au DEA informatique, la situation est toujours aussi désolante : on consacrera des dizaines d'heures à l'étude de techniques mathématiques intéressantes en soi, mais qui n'apportent rien de fondamentalement nouveau par rapport à l'enseignement des classes préparatoires, alors que la logique, domaine nouveau et formateur, est simplement absente.

2. D'autre part, alors que l'université a rapidement profité de la qualité de ses chercheurs-enseignants, les écoles d'ingénieurs sont placées sous la tutelle d'ingénieurs qui n'ont eux-mêmes qu'une mauvaise connaissance de ce qu'est devenue l'informatique[3] et comprennent mal la nécessité d'une évolution. La boucle est ainsi bouclée, et peut se révéler extrêmement dangereuse dans le système français qui donne aux ingénieurs des postes de décideur, et de responsable du développement dans les entreprises.

3. Enfin, la formation en informatique faite « à la base » dans les classes préparatoires tient plus du cours de serrurerie que d'un véritable cours scientifique, et les travaux pratiques de chimie font figure à côté de science fondamentale. L'informatique doit être enseignée par des gens formés à cet effet, et non par des amateurs, quelles que soient leurs qualités intellectuelles et leur bonne volonté.

Nous sommes persuadés qu'une meilleure compréhension et une meilleure utilisation de l'informatique dans la société passe avant tout par une vraie formation rigoureuse de tous les individus à tous les niveaux. Nous souhaitons rappeler à ce propos la phrase de Robert Escarpit (Escarpit 1981) :

> « Les craintes qui se manifestent devant l'extension et la multiplication des fichiers informatiques, et qui traduisent le refus individuel de la standardisation, sont irraisonnées et probablement exagérées. Le danger n'est pas dans l'informatique elle-même, qui est un excellent instrument pour saisir l'information, la traiter et même la produire par rapport au temps.[...] Ce que l'individu doit craindre, ce n'est certes pas de devenir transparent du fait de l'emploi de l'informatique, c'est d'être traité comme s'il était transparent, comme s'il était entièrement définissable et quantifiable par un nombre fini de paramètres. »

---

2 De grands logiciens français, comme Jean Cavaillès (Cavaillès 1962), sont morts jeunes, sans avoir pu réellement fonder une école. Suivant la phrase de Roger Martin (Martin-Lof 1966) : « [La France est un pays] où la recherche logique semble poursuivie par une perpétuelle malchance. ».

3 Il ne s'agit bien entendu pas d'un phénomène général. Il existe quelques exemples d'écoles ayant développé d'excellentes filières d'ingénieurs informaticiens.

Un individu instruit de la réalité des choses sera mieux à même de maîtriser ce nouveau domaine, et moins enclin à se laisser entraîner vers les images d'Epinal de l'informatique véhiculées par les médias.

Husserl écrivait (Husserl 1959), il y a presque un siècle, une phrase qui s'applique parfaitement à l'intelligence artificielle :

> « Incomparablement plus dangereuse est, par contre, une autre imperfection dans la délimitation du domaine d'une science, c'est la *confusion* des domaines, le mélange de choses hétérogènes et leur réunion en l'unité prétendue d'un seul domaine... [elle] peut, si l'on ne s'en aperçoit pas, avoir les effets les plus néfastes : la fixation de buts non fondés, l'observation de méthodes absurdes dans leur principe... »

Nous sommes conscients que cet ouvrage est critiquable : il couvre trop de domaines pour espérer présenter plus que les éléments essentiels de chacun d'entre eux, et pas suffisamment pour être une description « exhaustive » de l'informatique théorique ou de l'intelligence artificielle. Nous espérons tout de même qu'il vous permettra, de découvrir, d'apprécier, voire même d'approfondir (via les nombreuses références bibliographiques) quelques-uns des domaines, théoriques ou pratiques, à notre avis tous fascinants, qui y sont exposés. Nous espérons ainsi fournir des éléments permettant de mieux comprendre quelle est la place de l'intelligence artificielle vis-à-vis de l'informatique théorique et de faire, un peu, la part des choses dans les discours contradictoires que l'on entend sur ce sujet depuis bientôt trente ans. Si tel est bien le cas, alors ce travail n'aura pas été tout à fait vain.

Cette troisième édition profite de la participation de deux nouveaux auteurs : Pascal Brisset (ENAC) et Frédérick Garcia (INRA). Grâce au premier, le chapitre dédié à la programmation fonctionnelle, qui s'appuyait jusqu'ici sur le langage SCHEME, décrit maintenant le langage CAML, utilisant un typage inféré. Grâce au second, les méthode d'apprentissage par renforcement et les processus décisionnels de Markov font leur entrée dans la section « Apprentissage ». Naturellement, cette édition corrige également quelques erreurs et omissions. Le chapitre sur les réseaux de neurones a été remanié, les chapitres sur les CSP et les algorithmes génétiques mis à jour. Nous invitons le lecteur à nous communiquer ses remarques constructives par courrier électronique à l'adresse IAetIT@eis.enac.dgac.fr.

Toulouse, 28 février 2002

J.M. Alliot, P. Brisset, F. Garcia, T. Schiex,

CHAPITRE PREMIER

Définir l'Intelligence Artificielle

Jean-Marc Alliot

Un biologiste, un architecte et un chercheur en intelligence artificielle discutaient pour savoir laquelle de leurs professions était la plus ancienne. « La biologie est la plus ancienne, » prétendait le biologiste, « quand Dieu a créé Eve à partir de la côte d'Adam, ce fut là l'acte d'un biologiste. » « Certainement pas, » répondit l'architecte, « l'architecture est la plus ancienne. Quand Dieu a créé le monde à partir du chaos, ce fut là l'acte d'un architecte. » « Moi, » dit le chercheur en IA, « je pense que l'intelligence artificielle est la plus ancienne : d'où venait le chaos, d'après vous ? »

1.1 Qu'est-ce que l'intelligence artificielle ?

Il est extrêmement difficile de donner de l'intelligence artificielle une définition précise, tellement les domaines que recouvre le terme sont vastes et les avis divergents.

En fait, le terme même « intelligence artificielle » est considéré par certains auteurs comme parfaitement indu, voire contradictoire.

Il est aujourd'hui peu de disciplines qui provoquent des prises de position aussi claires et aussi tranchées, autant favorables que violemment critiques, à la fois sur la définition du terme et de ce qu'il recouvre, que sur les buts, la façon de les atteindre et l'intérêt même de la discipline.

Si pour certains (Feigenbaum, Simon, Newell, Winston,...) l'intelligence artificielle apparaît comme la science de l'avenir, de nombreux commentateurs ont une opinion radicalement opposée ; tout récemment encore Jean-Jacques Duby, directeur scientifique d'IBM Europe et membre du Conseil National des Programmes, déclarait (Duby 1990) :

> « La part de l'intelligence artificielle dans l'enseignement est disproportionnée par rapport à son importance et à ses potentialités. Il existe plus de DEA en intelligence artificielle que de systèmes experts industriels, et dans les rares qui sont utilisés, la technique de programmation est plus importante que la sémantique ou la représentation des connaissances. Autant alors enseigner la bonne vieille programmation[1]. »

---

1 Il est parfois bon de rappeler que l'intelligence artificielle n'est qu'une partie de l'informatique, et que des mots aussi mystérieux que *non-déterminisme*, *logique floue*, ou *algorithmes génétiques* s'interprètent simplement par le passage

Avant d'aller plus avant dans la polémique essayons de faire une petite analyse historique de l'IA, qui nous éclairera peut-être sur la raison de la violence des réactions qu'elle a provoquées.

1.2 Historique

1.2.1 La préhistoire : 1945-1955

L'IA a débuté, même si elle ne portait pas encore ce nom là, après la seconde guerre mondiale. Le premier problème auquel se sont attaqués certains informaticiens était la traduction automatique. Ils s'étaient consacrés pendant la guerre aux problèmes de décryptage et pensaient pouvoir adapter des méthodes classiques aux problèmes de traduction du langage. Leur erreur fut probablement de faire preuve d'un peu trop d'optimisme et surtout de croire qu'il était possible de réaliser une traduction sans qu'il soit nécessaire de comprendre le sens d'un texte. Ils pensaient ainsi réussir à mettre au point un traducteur automatique dans les cinq ans (!) et le clamèrent bien haut. Très vite, ils durent déchanter ; le problème se révéla tellement ardu qu'il fut pratiquement abandonné sous sa forme initiale. Les enseignements que l'on en tira furent néanmoins profitables : on se rendit en particulier compte de l'importance considérable que revêt dans le raisonnement humain l'ensemble de connaissances non exprimées. Ce premier échec amena à se poser des questions sur la représentation à donner aux connaissances, ainsi qu'à la façon de les « extraire » d'un individu. On se rendit compte aussi de l'impossibilité d'exprimer *toutes* les connaissances et donc de la nécessité de les rédiger sous des formes génériques. Toutes ces remarques donnèrent naissance à certaines branches de l'IA telle qu'elle allait apparaître.

Cependant le résultat pratique était décevant[2].

1.2.2 Les débuts : 1955-1970

Dans les années 55-65, d'autres équipes de recherche s'intéressèrent à des problèmes que l'on allait commencer à appeler problèmes d'IA. La définition du mot date d'ailleurs de la conférence tenue au Dartmouth College (New Hampshire, USA) où il est employé pour la première fois par John McCarthy. Dès le commencement, McCarthy fonde l'IA sur le postulat mécaniste qui veut que toute activité intelligente soit modélisable et reproductible par une machine. Nous allons revenir sur ce postulat de façon plus précise.

---

d'une exécution séquentielle à une exécution arborescente, par la combinaison mathématique simple de coefficients numériques ou par une méthode d'optimisation dans un espace discret. Trop souvent, ces mots sont employés par de pseudo-spécialistes, comme certains faiseurs de faux miracles employaient *abracadabra* ou *alakazam*, dans le seul but d'impressionner une assistance trop inculte, trop crédule ou trop heureuse d'assister là à un événement d'une telle importance. Cette forme d'hallucination collective doit en effet être dénoncée avec force, et, dans cette acception, on ne peut que partager l'avis de Jean-Jacques Duby.

2 Utile cependant. Le Maître d'échecs Tchigorine disait : « Mes défaites m'ont plus appris que mes victoires. » Cette remarque s'applique bien souvent à la recherche. Avoir échoué, c'est aussi avoir examiné à fond un domaine et avoir trouvé qu'il fallait utiliser une autre approche ou une autre théorie, ce qui est en soi un résultat positif.

C'est à cette époque que Newell, Shaw et Simon commencent à publier leurs premiers articles et à réaliser leurs premiers programmes.

Le premier programme d'IA est sans doute le LOGIC THEORIST (1956) dont le but est la démonstration automatique de théorèmes.

Ils développent conjointement un système de résolution de problèmes généraux (GPS : General Problem Solver) basé sur des techniques dites *analyse moyen-fin* et un programme de jeu d'échecs, NSS (1957).

Les structures des programmes sont relativement similaires. Elles tendent à reproduire le mieux possible le fonctionnement de l'esprit humain. Les idées mises en œuvre sont souvent intéressantes. C'est dans ces mêmes années qu'Arthur Samuel développe un programme de jeux de dames américaines (checkers). Ce programme de dames tentait d'améliorer son jeu en fonction de ses défaites passées par des techniques d'apprentissage[3].

Pour écrire facilement des programmes aptes à manipuler les informations symboliques, ils développent un langage, IPL1 (Information Processing Language), qui sera l'ancêtre de LISP développé par McCarthy en 1960. LISP va devenir le langage de l'IA pour les vingt années à venir. D'autres applications voient le jour, comme DENDRAL (qui peut être considéré comme un des premiers systèmes « experts »[4]), écrit à Stanford.

Simon et Newell sont les premiers à tenter de donner des fondements théoriques à l'IA. Leur postulat fondamental dit *postulat mécaniste* est que l'intelligence est une donnée générale des systèmes symboliques (Simon 1991). Voici leur texte ; ils définissent tout d'abord la notion de structure de symboles :

> « Une structure de symboles est composée d'un nombre d'occurrences (ou signes) de symboles reliés d'une manière physique. »

Puis ils expliquent ce qu'est un système de symboles physique :

> « Un système de symboles physique est une machine qui produit dans le temps un assemblage évolutif de structures de symboles. »

Enfin ils posent l'hypothèse fondatrice de l'IA :

> « Tout système de symboles physique possède les moyens nécessaires et suffisants pour une action intelligente de caractère général. »

En termes plus clairs, ce principe (dit PSSH en anglais *« Physical Symbol System Hypothesis »*, ou Hypothèse du Système de Symboles Physique), pose que :

> « Un système physique est capable d'un comportement intelligent (intelligent étant pris dans le sens d'*humainement* intelligent) ssi il s'agit d'un système de symboles physique. »

---

3 Notons tout de suite que les conclusions que Samuel d'une part et Simon, Newell, Shaw d'autre part, tirèrent de leur expérience sont fort différentes. Samuel est d'emblée relativement pessimiste, alors que l'équipe Newell, Simon, Shaw est résolument optimiste.

4 Il m'arrivera de mettre le mot expert entre guillemets en parlant des systèmes experts. Si l'on prend le mot *expert* dans le sens *résolution de problèmes dans un domaine particulier* alors les guillemets ne se justifient pas. En revanche si le mot *expert* se comprend *résolution comme le ferait un spécialiste de haut niveau du domaine* alors les guillemets s'imposent. Remarquons que la seconde définition reste cependant floue : *résoudre comme un spécialiste de haut niveau* signifie-t-il *résoudre et arriver au même résultat qu'un spécialiste* ou faut-il également imposer d'utiliser les mêmes moyens de résolution ? C'est la controverse entre pragmatistes et cognitifs sur laquelle nous allons revenir.

En un mot, l'esprit existe comme une implantation d'un système de symboles physique, et l'on doit pouvoir réaliser des systèmes de symboles physique de type programme de calculateur réalisant un comportement intelligent.

Ce principe est présenté par Simon et Newell comme une solution au problème « Comment l'esprit peut-il exister dans un univers physique ? » (Newell 1981). On voit que d'emblée Simon et Newell placent le problème sur un terrain mouvant. Leur définition « idéaliste » prête le flanc à la critique, comme nous le verrons plus loin.

C'est aussi l'époque de l'enthousiasme absolu ; Simon prédit ainsi en 1958 qu'un programme d'échecs arrivera au niveau d'un champion du monde dans les dix années à venir, et qu'un programme de démonstration automatique de théorèmes découvrira un théorème mathématique important dans la même période[5].

Ces exagérations feront, comme nous allons le voir, beaucoup de mal à l'IA.

1.2.3 La spécialisation : 1970-1980

La fin des années 60 et le début des années 70 vont surtout voir l'IA se spécialiser et se théoriser.

J. A. Robinson (Robinson 1965) développe le *principe de résolution* qui sera à la base de la réalisation de PROLOG, et des systèmes de résolution automatique. On commence alors à écrire les premiers systèmes à base de connaissances et de règles d'inférence.

En fait, l'IA va alors se scinder en plusieurs branches que l'on peut schématiquement résumer ainsi :

- la compréhension du langage naturel ;
- la démonstration automatique de théorèmes ;
- les jeux ;
- la résolution de problèmes généraux ;
- la résolution de problèmes experts ;
- la représentation de connaissances ;
- la perception ;
- l'apprentissage ;
- de façon générale tout problème que l'on ne sait pas résoudre informatiquement de façon classique[6].

Nous étudierons au fil de cet ouvrage certaines de ces disciplines, qui restent aujourd'hui les disciplines fondamentales de l'IA, ainsi que différentes techniques utilisées pour les aborder.

1.2.4 Une reconnaissance : 1980-1990

L'événement majeur des années 80 est l'arrivée en force des Japonais dans le domaine de l'intelligence artificielle. Le Japon lance en effet le projet *Cinquième génération* dont le

---

5 Il fit également une troisième prédiction, celle que d'ici dix ans, « la plupart des théories psychologiques revêtiront la forme de programmes d'ordinateurs, ou de commentaires sur les traits saillants de programmes d'ordinateurs ». Des trois prédictions de Simon, la dernière est sans doute la plus proche de la réalité, comme le fait très justement remarquer Dreyfus. Ce qui lui fait (perfidement) dire (Dreyfus 1984) : « Le seul intérêt de la simulation cognitive sera probablement de montrer que les êtres humains ne raisonnent pas comme des programmes d'ordinateurs ».

6 Cette dernière définition n'est pas acceptée par tous. Les cognitifs sont plus restrictifs que les pragmatistes. Voir paragraphe 1.3.

but est de développer, à la fois sur le plan matériel et sur le plan logiciel, des technologies et des techniques capables de faire de l'IA une discipline « efficace »[7].

Le projet *Cinquième génération* s'est officiellement terminé lors de la conférence internationale FGCS au mois de Juin 1992. Peut-on chiffrer les résultats de ce projet ? C'est relativement difficile ; sur un plan purement technique, il est clair que des progrès ont été faits sur les architectures parallèles de calculateur (Taki 1992; Kumon *et al.* 1992; Nakashima *et al.* 1992). Il y a eu également d'intéressantes avancées dans le domaine des langages de programmation logique (Hirata *et al.* 1992) et des systèmes d'exploitation (Itoh *et al.* 1992; Yashiro *et al.* 1992). Au niveau des applications, les résultats sont plus mitigés. Peut-être attendait-on trop du programme japonais : la presse a démesurément exagéré les objectifs du projet, et l'a ensuite, par un effet de balancier qui lui est coutumier, littéralement assassiné. Il y a pourtant d'intéressantes réalisations concernant les bases de données (Kawamura *et al.* 1992; Yasukawa *et al.* 1992), le traitement des informations génétiques (Ishikawa and others 1992; Hirosawa and others 1992; Yoshida *et al.* 1992; Tanaka 1992)...

Sur le plan « politique », les retombées du programme japonais sont immenses. Elles ont donné à l'IA une crédibilité et une audience qu'elle n'aurait probablement jamais obtenues. Les retombées sont également importantes sur le plan « économique » : les efforts financiers réalisés par les Japonais (500 millions de dollars de 1982 à 1990) ont encouragé américains et européens à se lancer dans la bataille en y engageant des moyens importants, dont profitent actuellement la plupart des chercheurs d'IA dans le monde. Ainsi le DARPA (Defense Advance Research Project Agency) a lancé le programme *Strategic Computer Initiative* en investissant 1 milliard de dollars de 1983 à 1990.

Bien sûr, toutes les entreprises n'ont pas adopté l'IA, IBM restant même un adversaire convaincu[8], mais Digital Equipment Corporation a développé quelques systèmes d'IA, dont un système expert de diagnostic de pannes informatiques, et beaucoup de dirigeants d'entreprise tentent de développer des approches IA dans des domaines les concernant[9].

---

7 Il existe une explication originale de l'intérêt des Japonais pour l'IA. Les Japonais possèdent trois alphabets ; deux d'entre eux, le Katakana et l'Hiragana, sont assez proches, dans leur principe, de l'alphabet européen, chacun des symboles ayant une valeur syllabique. Le Katakana était initialement un alphabet utilisé par les hommes. Il ne sert aujourd'hui qu'à la transcription de termes occidentaux importés récemment (exemple : télévision (*terebi*) est écrit en Katakana). L'Hiragana n'était utilisé que par les femmes et sert aujourd'hui pour l'écriture syllabique. On peut tout écrire en Hiragana si l'on ne connaît pas les idéogrammes Kanji. L'Hiragana et le Katakana sont des alphabets d'invention japonaise (ils sont inconnus des Coréens et des Chinois) relativement récente.

Le troisième alphabet japonais, le Kanji, est au contraire constitué d'idéogrammes, chaque idéogramme pouvant être considéré comme un résumé d'un ensemble de mots permettant d'exposer une notion complexe de façon immédiate. Cet alphabet évolue et possède des milliers de symboles différents dont 1300 au moins couramment utilisés. Le Kanji est le premier alphabet connu des Japonais et a été importé par les Chinois.

L'arrivée des ordinateurs obligeait les Japonais à l'abandonner. En se lançant dans les projets d'IA, certains pensent que les Japonais cherchaient à trouver des méthodes pour conserver le Kanji en le rendant compréhensible à des ordinateurs. Voir (Unger 1987) pour plus de détails.

8 Attitude générale au niveau du management chez IBM ; l'entreprise possède d'excellents chercheurs mais préfère ne pas bousculer un marché par l'introduction de techniques nouvelles. Pour une discussion plus précise sur IBM et l'IA voir (Feigenbaum and McCorduck 1984).

9 Souvent hélas sans savoir vraiment de quoi il s'agit, ni quel est le type de problèmes que l'on peut essayer de traiter ainsi, ni quelles sont les méthodes à utiliser pour mettre en œuvre ce genre d'approche. Il s'agit plutôt là d'un phénomène que l'on pourrait qualifier de « mode ». Ce type de phénomène a fait du tort à l'IA comme le fait très justement remarquer (Farreny and Ghallab 1987).

Il faut cependant remarquer que les années 80 ont vu certains objectifs de l'IA réalisés, en particulier, un ordinateur d'échecs (DEEP THOUGHT) a obtenu un rang proche de celui de Grand Maître International ; un certain nombre de systèmes experts à caractère opérationnel voient le jour ou s'affirment (PROSPECTOR, R1/XCON...).

1.3 Les différentes approches de l'IA

Nous venons de voir que l'IA a, au cours de son développement, abordé de nombreux sujets et connu bien des évolutions. On peut dire qu'actuellement deux conceptions relativement différentes se sont dégagées, l'une plus proche des sciences humaines et de la psychologie, l'autre qui considère davantage l'IA comme une science de l'ingénieur.

1.3.1 L'approche cognitive

L'approche cognitive[10], qui redevient très à la mode actuellement, après avoir eu ses heures de gloire au début des années 70, est essentiellement pluri-disciplinaire et l'on voit linguistes, informaticiens et psychologues essayer de travailler côte à côte.

William J. Rapaport la définit ainsi (Rapaport 1983) :

> « La science cognitive en général cherche à comprendre les fonctions cognitives humaines en termes d'états mentaux et de processus *i.e.*, en termes d'algorithmes[11] qui réalisent la transformation des données d'entrée en données de sortie... L'approche calculatoire de la science cognitive dit que : (1) il existe des états mentaux et des processus qui interviennent entre les stimuli d'entrée et les réponses en sortie, (2) ces états mentaux et ces processus sont des algorithmes (forme forte de la science cognitive[12]) ou en tous cas sont semblables à des algorithmes (forme faible) et, donc, (3) ces états mentaux et ces processus sont susceptibles d'être étudiés scientifiquement. »

Pour les défenseurs de cette thèse, l'IA est *la réalisation de programmes imitant dans leur fonctionnement l'esprit humain.* L'étude des processus des raisonnements humains est dans ce cas une fin en soi[13].

D'après Patrick Winston (Winston 1984) :

> « Apprendre aux ordinateurs à être plus intelligents permettra sans doute d'apprendre à l'homme à être plus intelligent. »

---

10 Certains, comme Searle, parlent d'*IA forte* et d'*IA faible.* Ces termes recouvrent à peu près nos définitions d'*approche cognitive* et d'*approche pragmatiste*, même si ces deux termes se rapportent plutôt à une notion de moyens, alors que Searle s'intéresse plus à la finalité.

11 Un algorithme est en informatique une description d'un processus de calcul totalement déterminé.

12 Ainsi, pour Pylyshin (Pylyshyn 1984) : « La cognition est une forme de calcul ».

13 Il est bon de faire une distinction entre l'approche cognitive et l'approche behavioriste. Pour les behavioristes, il est anti-scientifique d'étudier les processus de raisonnement, car seul ce qui est observable (*i.e.*, les comportements) peut-être objectivement étudié. Pour les cognitifs au contraire, c'est l'introspection et l'analyse de l'intuition qu'a le sujet de son raisonnement qui sont les domaines à étudier. La controverse entre behavioriste et cognitif a longtemps fait rage, surtout en linguistique. À l'intérieur même du monde cognitif, il existe d'ailleurs diverses écoles, comme le montre bien William Rapaport. Ce que nous désignons dans les lignes suivantes sous le nom de cognitif est plutôt appelé aux USA « computational cognitivism » or « the strong view of cognitivism ». Les partisans de l'approche cognitive faible, quant à eux, reconnaissent que les « états mentaux » n'existent pas forcément mais sont seulement un moyen commode de modéliser un comportement.

Ce type d'approche est généralement utilisé face à des problèmes trop complexes pour être abordés par d'autres voies, ou pour tenter d'éclairer les problèmes de fiabilité des systèmes reposant essentiellement sur des opérateurs humains, en les simulant au mieux, ou de procurer des aides à la décision aussi compréhensibles que possible pour l'expert humain, car ces systèmes sont sensés se rapprocher autant qu'il est possible du raisonnement d'un individu.

On trouve en particulier comme principaux défenseurs de cette approche les spécialistes de la compréhension du langage naturel[14].

1.3.2 L'approche pragmatiste

Pour les pragmatistes, les enseignements apportés par l'IA ne sont pas des fins en soi, mais des moyens pour développer des théories permettant d'améliorer notre capacité à programmer « efficacement » un ordinateur : le but du pragmatiste est de dégager de l'étude du problème des *algorithmes*, en tenant compte des contraintes imposées par la structure de l'ordinateur. On cherche donc à utiliser la machine au mieux de ses capacités et si possible à obtenir de meilleurs résultats que ceux que pourrait obtenir un être humain.

Pour un pragmatiste, une « intelligence artificielle » peut être considérée comme une boîte noire où rentreraient d'un côté les données du problème et sortiraient de l'autre coté les résultats, sans que l'on se préoccupe vraiment de ce qui se passe dans la boîte. La boîte serait considérée comme intelligente à compter du moment où elle réussirait un certain nombre de tests, qui, réussis par un humain, permettraient de dire qu'il est « intelligent » (au moins pour le domaine concerné). On peut rapprocher cette définition de celle que proposait Alan Mathison Turing dans un texte célèbre où il définit ce qui sera appelé le *test de Turing* (Turing 1983) :

> « Ce nouveau type de problème peut être présenté sous forme d'un jeu, que nous appellerons « jeu d'imitation ». Il nécessite trois joueurs : un homme (A), une femme (B) et un troisième joueur qui posera les questions (C), et qui peut appartenir à l'un ou l'autre sexe. Ce dernier restera dans une pièce séparée de celle où se trouve le couple. Le but du jeu, pour ce troisième joueur, est de déterminer, chez les deux autres joueurs, qui est l'homme et qui est la femme. Il les connaît sous le nom de X et de Y et doit dire à la fin du jeu, soit « X est A et Y est B », soit « X est B et Y est A ». Pour ce faire, il a droit à des questions du type : « X peut-il me dire quelle est la longueur de ses cheveux ? »
>
> « Supposons que X est en fait A, c'est à X de répondre. La règle du jeu pour A, c'est de tenter d'induire C en erreur. Par conséquent, il pourra répondre : « J'ai les cheveux coupés au carré, les plus longues mèches atteignent vingt-deux centimètres ». Pour éviter, naturellement, que les voix n'aident C, les réponses devront être écrites, si possible à la machine. Le mieux serait un

14 De toute évidence, la compréhension du langage naturel ne peut s'appréhender que par des méthodes cognitives. En effet, s'il est possible de construire un modèle mathématique pour le jeu d'échecs ou même la synthèse chimique, le langage est le résultat d'une activité purement humaine, qui n'a pas de réalité extérieure à l'homme : si l'homme n'existait pas, on pourrait toujours mélanger de l'oxygène et de l'hydrogène, en revanche le langage n'existerait pas. Cela pourrait d'ailleurs, si l'on était pessimiste, condamner tout espoir de voir un jour un ordinateur « comprendre » le langage naturel.

> télé-imprimeur dans les deux pièces ; ou encore, autre solution, les questions et les réponses peuvent être transmises par un intermédiaire. Le but du jeu pour la joueuse (B) est d'aider le joueur (C). La meilleure tactique en ce qui la concerne est peut-être de ne donner que des réponses justes. Elle peut ajouter des déclarations telles que : « c'est moi la femme, ne l'écoutez pas ! », mais comme le joueur A peut également tenir pareil discours, cela n'avancera pas à grand chose.
>
> « Et nous posons maintenant cette question : « Que se passera-t-il si l'on fait tenir par une machine le rôle de A dans ce jeu ? » Le joueur C, en pareil cas, échouera-t-il aussi souvent que lorsque le jeu est joué avec un homme et une femme pour partenaires ? Ces questions prennent la place de notre interrogation première : « Ces machines sont-elles capables de penser ? »

Ainsi, dans le cadre de cette approche pragmatiste, un grand nombre de techniques d'IA dont nous parlerons dans les chapitres suivants (espace d'états, parcours d'arbres et de graphes, algorithmes minimax et α-β) modélisent assez peu, à notre avis, le fonctionnement d'un esprit humain. Il est cependant assez difficile de les exclure historiquement de l'IA, comme le font certains tenants de l'approche cognitive, d'autant que ces domaines sont parmi les rares à avoir donné des résultats concrets.

Se réclament de cette approche de nombreux systèmes « experts[15] » utilisés à l'heure actuelle, les programmes de jeux les plus efficaces, ainsi que nombre d'autres programmes dans un peu tous les domaines de l'IA.

Enfin, il est clair que dans cette approche, la valeur du mot « penser » est extrêmement faible. Certains programmes sont réputés avoir passé le test de Turing (dont PARRY de Kenneth Colby qui simule un être humain paranoïaque (Colby *et al.* 1971; 1972)), sans que personne n'accepte de dire que le programme « pense » à proprement parler. Ainsi, Joseph Weizenbaum dans (Weizenbaum 1984), pourtant auteur du programme ELIZA qui simule un psychiatre de façon relativement convaincante, s'oppose violemment à toute interprétation des résultats de son programme comme un exemple de « pensée artificielle ».

1.3.3 L'approche connexionniste

Très tôt, nombre de chercheurs ont remarqué le paradoxe suivant : alors que les systèmes symboliques sont censés se montrer efficaces sur des activités typiquement humaines, ils n'ont donné des résultats véritablement tangibles que sur des domaines où l'être humain a besoin d'apprendre, alors que sur des domaines apparaissant comme innés (le langage, la marche) ils se sont montrés parfaitement inefficaces. Ceci a renforcé la critique des pragmatistes vis à vis de l'approche cognitive, et a amené certains chercheurs à un autre type d'approche : l'approche connexionniste.

L'approche connexionniste se place complètement en marge de tout ce que nous avons vu jusque là. Il ne s'agit plus de faire un apprentissage symbolique d'un processus humain, ni d'écrire un programme mimétique du raisonnement humain. Il s'agit d'une modélisation très grossière, très imparfaite, et très réductrice du fonctionnement

---

15 Le problème avec les systèmes experts, nous y reviendrons, est qu'il est bien difficile de les classer. Feigenbaum parle *d'ingénierie des connaissances*, mais il est souvent peu clair de voir à quelle approche ils appartiennent vraiment.

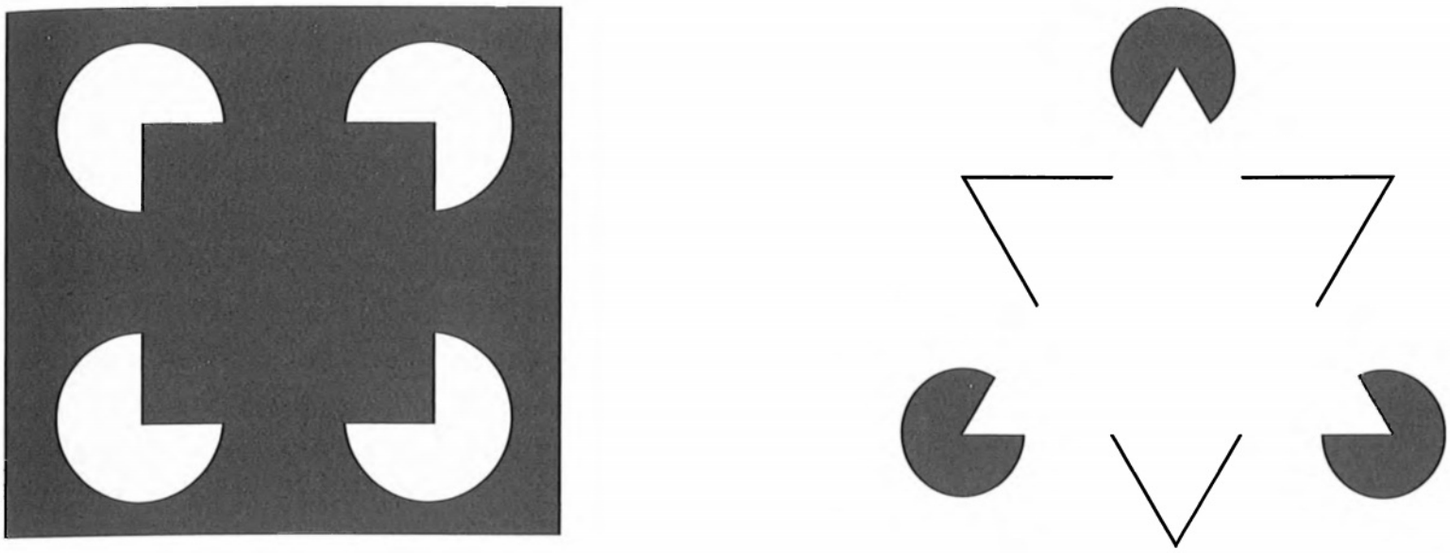


Figure 1.1 – Carré et triangle de Kanizsa

des neurones du cerveau humain. Nous reviendrons en détail dans le chapitre 20 sur les techniques des réseaux de neurones.

Disons grossièrement qu'un système connexionniste est composé de nœuds interconnectés. Ce système est susceptible d'apprendre automatiquement un certain nombre d'exemples, puis de généraliser (d'une façon empirique) cet apprentissage à d'autres échantillons, sans qu'il faille à aucun moment décrire une quelconque représentation symbolique de la connaissance, et sans que le système ne se préoccupe jamais de cet aspect du problème.

Il semble qu'à l'intérieur du cerveau humain, de nombreux mécanismes, même considérés comme complexes, sont effectués à un niveau directement mesurable suivant ce schéma. Une expérience récente (Dresp 1991) montre que certaines illusions visuelles comme les figures de Kanizsa[16] (voir figure 1.1) ont une représentation simple mesurable au niveau des neurones.

1.3.4 Convergence de ces approches ?

Contrairement à ce qu'écrivent certains auteurs (Farreny and Ghallab 1987), il ne semble pas que les approches pragmatistes et cognitives soient en passe de converger, bien au contraire.

Certains domaines reconnus comme étant de l'IA à leur début (les jeux par exemple) ont lentement intégré ou réintégré des domaines plus classiques (l'informatique traditionnelle, l'algorithmique...) et sont devenus la chasse gardée des pragmatistes, qui considèrent souvent les cognitifs plus comme des spécialistes de sciences humaines[17] que comme des scientifiques.

---

16 Ces figures présentent une illusion visuelle classique. Sur l'une, on « voit » un carré noir sur fond noir qui n'existe pas, sur la seconde on distingue un triangle blanc sur fond blanc.

17 Avec ce que cela implique souvent de péjoratif dans leur esprit.

D'autre part, les cognitifs estiment souvent être les seuls véritables héritiers de l'IA originelle et considèrent les pragmatistes comme des informaticiens ou des logiciens « traditionnels[18] ».

Certains auteurs estiment, en fait, que l'IA s'intéressant avant tout à ce que les ordinateurs savent mal faire aujourd'hui, une discipline classée comme étant de l'IA ou du domaine cognitif aujourd'hui ne le sera plus forcément demain. On pourrait alors voir les cognitifs comme les défricheurs d'un chemin que pourraient ensuite emprunter les pragmatistes, avant qu'il ne soit livré aux rouleaux compresseurs de l'industrie[19]. Cette vision des choses[20] ne satisferait probablement d'ailleurs ni les pragmatistes ni les cognitifs[21].

De façon générale une synthèse entre ces deux approches me semble aussi difficile à réaliser qu'une synthèse entre sciences sociales et sciences mathématiques. Les échanges entre les genres peuvent être fructueux, mais l'évolution n'apparaît pas comme convergente[22].

Quant à l'approche connexionniste, elle est résolument différente des deux précédentes. Le système est réduit à une « boîte noire » dont la structure interne, et à la limite, le fonctionnement même, sont inconnus. Il existe certaines tentatives pour utiliser parallèlement les techniques des réseaux de neurones et les techniques symboliques, mais il s'agit bien d'une utilisation parallèle, et non d'une utilisation convergente des deux domaines.

1.4 Notre définition de l'IA

Nous venons de faire un tour plus que rapide de l'IA, de sa progression et des critiques qui l'ont accompagnée, et continuent de l'accompagner. Comme nous l'avons vu, l'IA

---

18 Ce qui, là non plus, n'est pas dépourvu d'une certaine nuance péjorative.

19 On pourrait aussi dire que les cognitifs ont une approche « top-down » : ils partent de l'homme et tentent d'en faire un modèle que l'on fera fonctionner sur une machine ; en revanche, les pragmatistes ont une approche « bottom-up » : partant de techniques mathématiques et informatiques « lourdes », ils tentent de réaliser des programmes « intelligents ». Nous reviendrons sur tout cela dans le chapitre 23.

20 Marvin Minsky : « L'algèbre symbolique était de l'IA lorsque Seagle écrivit sa thèse en 1961. Les méthodes devinrent plus raffinées et plus sûres lorsque Moses rédigea la sienne en 1966 et l'approche heuristique avait presque complètement disparu lors du développement de MACSYMA, parce que les bases mathématiques avaient été largement développées (par Risch et Caviness en particulier). »

21 Il existe au moins un domaine qui a suivi cette voie, c'est la programmation du jeu d'échecs. Commencé en science cognitive avec la participation de psychologues, il devient ensuite le domaine privilégié du développement de la théorie de parcours d'arbres et des fonctions d'évaluation avant de passer dans le domaine industriel avec le marché des micro-ordinateurs d'échecs où, il faut bien le reconnaître, l'IA « forte » a complètement disparu.

22 Si l'on se plaçait dans une perspective d'histoire des sciences, et en particulier dans le cadre de l'analyse qu'en fait Thomas Kuhn (Kuhn 1983), on pourrait se demander si la branche « psychologique » de l'IA ne se trouve pas dans une période pré-paradigmatique, où plusieurs écoles donnent des interprétations différentes de la même masse chaotique d'observations, avant qu'une des écoles ne s'impose. Terry Winograd a écrit : « Il est peut-être trop tôt pour comparer l'état actuel de l'IA à la chimie moderne. En un certain sens, elle est plus proche de l'alchimie médiévale. Nous en sommes à l'étape où l'on mélange différentes substances et où l'on regarde ce qui se passe, sans avoir encore développé des théories satisfaisantes. Cette analogie fut proposée par Dreyfus en 1965 comme une condamnation de l'IA, mais le fait qu'elle soit correcte n'implique pas forcément qu'elle soit négative. Bien sûr, de nombreux travaux peuvent être critiqués car ils sont esclaves du but consistant à créer de l'or (l'intelligence) à partir de plomb (les ordinateurs). Pourtant, c'est l'expérience pratique et la curiosité des alchimistes qui ont fourni l'ensemble de données à partir desquelles la chimie moderne a pu être développée. » Les paroles de Winograd semblent parfaitement s'appliquer à la branche cognitive de l'IA, et sont, finalement, fort cruelles : elles condamnent sans rémission les méthodes de l'IA cognitive pour n'accorder de l'importance qu'aux données qu'elle recueille.

s'intéresse à nombre de problèmes très différents et les approches adoptées sont multiples et parfois divergentes. On pourrait, suivant le point de vue choisi, donner des définitions différentes de l'IA. Les cogniticiens parleraient probablement de *modélisation aussi fidèle que possible du raisonnement humain*, les pragmatistes diraient qu'il s'agit de définir des systèmes *manifestant les mêmes symptômes d'intelligence qu'un être humain.* Il nous faut essayer de trouver un dénominateur commun à toutes ces définitions. En ce qui nous concerne nous prendrons comme définition :

Définition 1.1 – Intelligence artificielle – *l'intelligence artificielle a pour but de faire exécuter par l'ordinateur des tâches*[23] *pour lesquelles l'homme, dans un contexte donné, est aujourd'hui meilleur*[24] *que la machine.*

Cette définition est adoptée par de nombreux auteurs[25] et englobe à peu près les définitions pragmatistes et cognitives, même si elle apparaîtrait comme peu acceptable aux tenants purs et durs de ces thèses.

C'est en tout cas la définition à laquelle nous nous référerons tout au long de ce livre, et c'est dans ce cadre que nous allons présenter un certain nombre de problèmes classiques d'IA. Remarquons tout de suite son caractère mouvant. Si demain une machine devient capable de mieux résoudre un problème qu'un être humain, ce problème cessera d'être un problème d'IA. D'autre part, la résolution de certains problèmes que la majorité acceptera comme faisant partie des problèmes mieux résolus par l'homme que par la machine (preuve de théorèmes par exemple) se ramène souvent à la résolution d'autres problèmes que beaucoup n'accepteront pas comme faisant partie des problèmes mieux résolus par l'homme que par la machine[26].

1.5 Problèmes typiques d'IA

Comme nous venons de le voir, l'IA doit surtout être appliquée à des problèmes que nous ne saurions pas résoudre autrement avec un ordinateur. Dans ce chapitre, nous allons détailler certains problèmes typiques (et presque caricaturaux) que les techniques

---

23 On parle surtout de tâches intellectuelles, évidemment, quoique la robotique s'intéresse aussi à des tâches plus « physiques ».

24 Avec le flou que cela sous-entend.

25 Elle aurait été donnée pour la première fois par McCarthy, et reprise par Rich. Laurière (Laurière 1986) a une définition peu différente : « Tout problème pour lequel n'existe aucune solution algorithmique connue est un problème d'IA ». Puis il définit « algorithmique » comme une suite d'opérations définies et ordonnées exécutables en un temps raisonnable.

On peut également citer la définition de Drew McDermott et Eugene Charniak : « l'intelligence artificielle est l'étude des facultés mentales à l'aide de modèles de type calculatoire ».

Rappelons celle de Patrick Winston : « l'intelligence artificielle est l'étude des idées qui permettent aux ordinateurs d'être intelligents ».

Terminons par celle de Edward Feigenbaum : « l'intelligence artificielle est une méthodologie qui doit permettre de rendre les ordinateurs plus intelligents de façon à ce qu'ils montrent des caractéristiques normalement associées à l'intelligence dans les comportements humains, c'est-à-dire la compréhension du langage, l'apprentissage, la résolution de problèmes, le raisonnement... »

26 Le problème de preuve de théorème en logique propositionnelle se ramène à une preuve d'inconsistance qui peut elle-même se ramener à un problème de programmation linéaire en variables bivalentes (cf. (Hooker 1988) par exemple), problème de recherche opérationnelle s'il en est. Quand et où sommes-nous sortis (si c'est le cas) du domaine de l'IA ?

de l'IA peuvent aider à étudier. Nous emploierons certains mots, mis en italiques, dont les définitions précises seront données dans les chapitres suivants, mais qui sont placés volontairement dans les énoncés des problèmes afin de leur donner un sens intuitif immédiat.

1.5.1 Les problèmes spécifiables et difficiles

Nous allons tout d'abord présenter une première catégorie de problèmes : ceux que nous savons totalement spécifier mais dont nous savons que la résolution n'est pas raisonnablement envisageable (voire impossible) par des méthodes algorithmiques existantes[27]. Voici trois exemples illustrant ce type de problèmes :

Le problème de Post : Développons sur un exemple le problème de Post que nous aurons l'occasion de revoir dans le chapitre consacré à la calculabilité. Soit deux triplets (abb, ab, aba) et (bb, aba, ba). Existe-t-il une liste d'indices telle que la concaténation des éléments indicés du premier triplet donne le même « mot » que la concaténation des éléments correspondant aux mêmes indices pour le second triplet ? Dans le cas présent, la réponse est oui de façon claire, il suffit de prendre la liste $(2, 3, 2, 1)$ qui donne pour les deux triplets *ababaababb*. Dans le cas général, ce problème est néanmoins *indécidable* : il n'existe pas d'algorithme permettant de le résoudre sur n'importe quelle instance dans un temps toujours fini. Supposons que nous ayons cependant à programmer un algorithme tentant de fournir une solution à ce problème ; il est clair que l'utilisation de la *force brute* qui consisterait à tester méthodiquement toutes les solutions n'aboutirait généralement pas. Il nous faut donc un moyen de diriger la recherche de l'ordinateur de façon « intelligente » ; pour ce faire nous pouvons interroger un *expert* du problème (un mathématicien par exemple), ou utiliser toute méthode permettant de dégager des *heuristiques*. Une heuristique permet d'améliorer (en moyenne) la capacité de l'ordinateur à résoudre le problème en le guidant dans l'espace des états à examiner. Nous pourrons aussi éliminer certains problèmes dont nous savons qu'ils ne sont pas solubles (par exemple lorsque les éléments du premier triplet commencent par des lettres qui sont toutes différentes des premières lettres des éléments du second triplet) ; nous pourrons ainsi coder des règles permettant d'élaguer l'*arbre de recherche* sans qu'il perde ses bonnes propriétés. Nous pourrons enfin inclure des règles nous permettant d'arrêter la recherche au bout d'un certain nombre d'opérations car nous estimerons alors que la réponse est probablement négative. L'arrêt de l'algorithme n'assure plus alors que le problème est résolu.

Le problème du voyageur de commerce : Il s'agit là de construire un programme permettant de trouver le plus court chemin passant par n villes données, sans jamais repasser par la même ville. Nous verrons (théorie de la complexité) que ce problème est NP-complet *i.e.*, il n'existe presque certainement pas d'algorithme permettant de le résoudre efficacement[28]. Là encore, une approche raisonnable pourrait consis-

---

27 Faut-il vraiment considérer ces problèmes comme des problèmes d'IA ? Oui, d'après la définition que nous avons prise, mais cela est un sujet de controverse.

28 Il y a là aussi un problème. En effet, la seule chose que nous puissions démontrer est qu'un calculateur *numérique* ne peut résoudre le problème en un temps « raisonnable ». En revanche, un calculateur *analogique* peut parfaitement y parvenir. L'homme est-il un calculateur numérique ?

ter à demander à un expert (un voyageur de commerce ?) sa méthode pour résoudre le problème. Il pourra par exemple nous conseiller de choisir systématiquement comme ville suivante la ville la plus proche parmi celles que nous n'avons pas visitées. Notre programme sera alors en mesure de nous donner une « solution » approximative dans un temps raisonnable (polynomial). Bien entendu, cette solution ne sera pas toujours la meilleure, et nous pouvons même démontrer qu'il est impossible de majorer l'erreur que nous commettons en appliquant quelque algorithme efficace que ce soit pour résoudre le problème (théorie de la complexité).

Les échecs : Le jeu d'échecs, comme la plupart des jeux, fait partie du patrimoine de l'IA depuis les débuts de celle-ci. Un des premiers programmes d'IA fut d'ailleurs un programme de jeu, NSS, réalisé par Newell, Simon et Shaw, ainsi que le programme de dames de Samuel qui utilisait les résultats de ses précédentes parties pour « apprendre » et améliorer son jeu. Nous reviendrons dans un chapitre spécifique sur le problème de la programmation des jeux. Soulignons simplement là aussi que les techniques d'IA ont été mises en œuvre en raison de la grande complexité du problème : il est clairement impossible de faire trouver au programme la suite de coups menant au mat dès l'ouverture.

1.5.2 Problèmes recouvrant un domaine précis

La deuxième classe de problèmes à laquelle nous allons nous intéresser correspond à des problèmes moins clairement spécifiables, mais cependant relativement bien cernés, et définis dans ce que l'on a coutume d'appeler un *micro-monde* et dont les buts sont clairs. Ce type de problème est souvent lié à la notion de *systèmes experts*. Voici quelques exemples :

MYCIN : MYCIN a été développé dans les années 60. Il s'agit d'un système expert de diagnostic médical. On fournit à la machine un ensemble de symptômes et en fonction de ces symptômes, le programme tente de formuler un diagnostic. Le problème est plus vaste que les précédents, les données et les règles sont susceptibles d'évoluer avec le temps. Le fonctionnement de MYCIN est considéré comme satisfaisant. Il faut cependant remarquer que dans ce cas également les techniques utilisées dans le programme sont fort peu proches d'un raisonnement humain classique (un praticien utilise rarement des coefficients numériques de croyance pour établir un diagnostic).

DENDRAL : est un système d'analyse chimique. Nous le présenterons plus en détail dans le chapitre 16.

PROSPECTOR : système expert d'aide à la prospection géologique.

R1/XCON : Système expert développé par Digital Equipment Corporation permettant, en fonction des contraintes données par un client (moyens financiers, espace disponible, mémoire requise, puissance de calcul désirée), de calculer la configuration optimale du système informatique à fournir. Ce système comprend plusieurs milliers de règles et est opérationnel (et utilisé).

De nombreux autres exemples de systèmes experts pourraient être évoqués ici, le système expert étant devenu une quasi-réalité industrielle. Rappelons une fois de plus cependant que nombre de systèmes experts n'ont d'expert que le nom et que les techniques utilisées pour les développer sont souvent bien éloignées des techniques d'IA. Nous éviterons de

parler de ce type de système, mais il faut signaler cette tendance actuelle qui a pour but de vendre un logiciel en lui accolant le terme *expert* et le qualificatif *intelligent*.

1.5.3 Autres domaines liés à l'IA

L'IA ne se limite pas aux différents exemples cités ci-dessus. Il est classique d'inclure également dans les problèmes d'IA :

- la résolution de problèmes de façon générale[29] ;
- la compréhension du langage naturel (et non de mots isolés) ;
- la reconnaissance de scènes visuelles continues (et non d'images fixes) ;
- l'étude de la fiabilité des opérateurs humains ;
- l'apprentissage ;
- le modèle cognitif du raton-laveur, etc.

D'autre part, l'IA a des relations étroites avec des domaines proches de la recherche opérationnelle (parcours d'arbres et de graphes, par exemple), des mathématiques appliquées, de la logique, des mathématiques « pures », etc.

Il est ainsi tout aussi difficile de donner une liste exhaustive des problèmes adressés par l'IA qu'il est malaisé d'en donner une définition. Dans la suite de cet ouvrage, nous nous sommes efforcés de traiter les domaines qui nous paraissaient essentiels, sachant que tout choix est arbitraire et sujet à critique.

---

29 On dit que l'on utilise des techniques générales quand on emploie des techniques qui peuvent s'appliquer à n'importe quel type de problèmes. Le General Problem Solver (GPS) fut la première tentative de cet ordre.

Partie I

Logique mathématique, résolution

> Here, you see, it takes all the running you can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that !
>
> — *Lewis Carroll*

CHAPITRE 2

Le calcul propositionnel

Jean-Marc Alliot — Thomas Schiex

2.1 Méthodologie

L'approche que nous allons suivre au fil de ce chapitre constitue une technique générale pour étudier un système logique quelconque. Nous adopterons la même approche lorsque nous étudierons les logiques non-classiques. Nous allons tout d'abord établir le *langage* avec lequel nous allons opérer ; puis nous développerons une théorie des modèles, qui permettra de donner un sens aux formules de notre langage logique, que nous appellerons interprétation *sémantique*. Nous étudierons parallèlement une théorie formelle de la démonstration sur ce langage qui, sans s'intéresser au sens des formules, permettra de les manipuler. Cette théorie formelle correspond à l'étude axiomatique de la logique : elle ne s'intéresse qu'aux propriétés *syntaxiques*[1].

Enfin, nous étudierons les relations entre ces deux « visions » de notre logique afin de vérifier que les interprétations que nous avons adoptées sont bien cohérentes entre elles. L'étude des systèmes formels sera un peu différente dans la mesure où nous n'aurons pas recours aux modèles, et où nous ne nous intéresserons qu'à des propriétés formelles[2].

Il semble au premier abord que l'étude de la logique se heurte à un paradoxe de taille ; comment en effet étudier la logique sans avoir à recourir à la logique elle-même ? Pour résoudre ce problème, nous emploierons tout au long de ces chapitres deux langages : le *langage-objet* qui sera le langage du modèle que nous étudions, et le *langage de l'observateur* qui sera le langage dont nous nous servirons pour étudier le système logique. Ce

---

1 Nous ne pouvons que recommander la lecture de (Kleene 1987; Fraïssé 1971; Delahaye 1986; Gochet and Gribomont 1990; Lalement 1990) ou, pour une introduction très claire mais plus élémentaire (Rivenc 1989) (ouvrages en français).

2 Il est peut-être bon de réfléchir un peu plus profondément sur cette double vision de la logique. De façon intuitive (et historique), l'interprétation sémantique vient avant l'interprétation syntaxique. Nous sommes face à un ensemble de propriétés dont nous savons qu'elles sont vraies (par exemple si A est vrai et B est vrai, alors (A et B) est vrai). Notre problème est que bien souvent il est difficile de réaliser une interprétation sémantique pour des propositions plus complexes. Nous cherchons alors à construire des méthodes plus mécaniques, et donc moins sujettes à des erreurs de jugements, nous permettant de dire si une proposition est vraie ou fausse. Pour ce faire, il nous faut construire une théorie syntaxique de la démonstration, puis établir l'équivalence entre les deux approches (syntaxique et sémantique). C'est exactement le propos de la logique mathématique que de s'intéresser à ce type de problèmes. Les systèmes formels n'échappent pas à ce mécanisme général, nous y reviendrons au chapitre 5.

La logique philosophique s'intéresse surtout au sens « substantiel » de la vérité des modèles sémantiques, c'est-à-dire au rapport entre la théorie sémantique et la vérité philosophique que l'on peut accorder à cette théorie.

langage de l'observateur est parfois appelé *méta-logique* ou *méta-langue*. Cette distinction est absolument essentielle pour étudier la logique. Nous verrons que bon nombre de paradoxes ont leur origine dans une séparation insuffisante entre langage-objet et langage de l'observateur.

2.2 Introduction informelle

Le but initial de la logique est de formaliser certaines relations qui peuvent apparaître de bon sens. Si je dis la phrase p : « il fait beau et je suis en vacances », cette phrase sera vraie si les deux propositions q : « il fait beau » et r : « je suis en vacances » sont vraies toutes les deux. Si l'une des deux est fausse, alors p sera fausse. Nous utilisons donc une interprétation intuitive du *connecteur* **et** qui nous permet de déterminer la *valeur de vérité* d'une phrase contenant deux sous-propositions reliées par **et** si nous connaissons la valeur de vérité de chacune des propositions. De même, nous utilisons un mécanisme semblable pour des phrases contenant les connecteurs *ou*, *donc*... C'est le *principe de compositionalité*.

Le but du calcul propositionnel est de donner un fondement formel à un ensemble restreint d'énoncés du langage. Cet ensemble restreint n'utilise que quatre connecteurs (*et*, *ou*, *donc*, *équivalent à*), la négation (si la phrase « il fait beau » est vraie alors « il ne fait pas beau » est fausse) et a comme éléments de base des propositions élémentaires « il fait beau », « je suis en vacances » que les connecteurs vont relier entre elles pour former les phrases.

2.3 Définitions

Définition 2.1 – Atomes – *Nous appellerons* atomes *ou* variables propositionnelles *des énoncés dont nous ne connaissons pas (et ne cherchons pas à connaître) la structure interne, et qui gardent leur identité tout au long du calcul propositionnel qui nous occupe. Nous noterons ces atomes par les lettres minuscules de l'alphabet (généralement : p,q,r,s...). L'ensemble (infini) des variables propositionnelles est noté V_p.*

Définition 2.2 – Connecteurs – *Nous appellerons* connecteurs propositionnels *les symboles suivants :*

- $\leftrightarrow$ *équivalence*
- $\rightarrow$ *implication*
- $\wedge$ *conjonction*
- $\vee$ *disjonction*
- $\neg$ *négation*

Définition 2.3 – Formules – *Nous dénoterons les formules (ou formules bien formées) par les lettres majuscules de l'alphabet latin (A,B...). Elles sont définies par :*

- *les atomes sont des formules ;*
- *si A et B sont des formules alors $(A \rightarrow B)$, $(A \leftrightarrow B)$, $(A \wedge B)$, $(A \vee B)$ sont des formules ;*
- *si A est une formule, alors $(\neg A)$ est une formule.*

Toute formule[3] *est obtenue par l'application des règles précédentes un nombre fini de fois.*

Nous utiliserons aussi par la suite la formule notée $\varnothing$ et explicitement définie par $\varnothing =_{def} (a \wedge \neg a)$, où a est une variable propositionnelle quelconque.

Définition 2.4 – Littéral – *Un littéral est un atome (aussi appelé littéral positif) ou la négation d'un atome (aussi appelé littéral négatif).*

Une formule de la forme $p \leftrightarrow q \rightarrow r$ est ambiguë. S'agit-il de $((p \leftrightarrow q) \rightarrow r)$, c'est-à-dire de la formule $(p \leftrightarrow q)$ à laquelle nous avons connecté à droite r via $\rightarrow$ ou de $(p \leftrightarrow (q \rightarrow r))$, qui correspond à $(q \rightarrow r)$ auquel nous avons connecté à gauche p via $\leftrightarrow$? Les parenthèses sont donc un moyen permettant de lever l'ambiguïté. Il en existe un autre qui consiste à donner à chaque opérateur un ordre de dominance. Les opérateurs sont traditionnellement classés de la façon suivante : $\leftrightarrow$, $\rightarrow$, $\wedge$, $\vee$, $\neg$. On se permet alors d'omettre les parenthèses.

Par exemple, la formule : $p \rightarrow q \leftrightarrow \neg r$ doit se lire $((p \rightarrow q) \leftrightarrow (\neg r))$.

L'ensemble des formules bien formées ainsi défini forme le langage de la logique propositionnelle.

2.4 Théorie des modèles

2.4.1 Introduction

La théorie des modèles a pour but d'établir un mécanisme sémantique d'évaluation des formules. Par sémantique, nous entendons que nous allons donner un sens à nos atomes, puis donner un sens à une formule à partir de la valeur de vérité des atomes qui la composent et des connecteurs qui relient ces atomes. La théorie des modèles est une formalisation de la notion intuitive que nous avons de la vérité ou de la fausseté d'une phrase en fonction des propositions qui la composent.

En calcul propositionnel, ce résultat est obtenu en donnant à chaque atome une *valeur de vérité* et en associant à chaque connecteur une *table de vérité* qui, en fonction de la valeur de vérité des arguments du connecteur, donne la valeur de vérité de la formule constituée.

Notre hypothèse, *en calcul propositionnel*, est donc que chaque atome ne peut prendre que deux valeurs : vrai (t) ou faux (f), et que la valeur de vérité d'une composition de formules est entièrement déterminée par la valeur de chacun de ses arguments (ce ne sera pas le cas, nous le verrons plus loin, en logique modale par exemple).

2.4.2 Définition formelle en calcul propositionnel

Définition 2.5 – Valuation – *On appelle valuation d'un ensemble de variables propositionnelles $V \subset V_p$ une fonction de V dans $\{t, f\}$.*

---

3 De nombreux auteurs se limitent à l'utilisation de deux connecteurs ($\neg$ et $\vee$ par exemple) pour l'élaboration des formules, les autres connecteurs étant définis dans les termes de ces connecteurs élémentaires. Il est même possible de se limiter à un seul connecteur : le | ou barre de Sheffer ($A|B$ est égale à faux ssi A et B sont égaux à vrai) qui correspond aux portes NAND des circuits logiques. La barre de Sheffer duale peut aussi être utilisée, elle correspond à une porte NOR.

A	B	$(A \leftrightarrow B)$	$(A \rightarrow B)$	$(A \wedge B)$	$(A \vee B)$
t	t	t	t	t	t
t	f	f	f	f	t
f	t	f	t	f	t
f	f	t	t	f	f

Table 2.1 – Connecteurs dyadiques

Une valuation permet de connaître, pour chaque variable propositionnelle, sa valeur de vérité. Elle va permettre d'interpréter une formule :

Définition 2.6 – Interprétation d'une formule – *Une interprétation d'une formule F dans laquelle apparaissent les variables propositionnelles $v_1, \ldots, v_n$ est une valuation de $\{v_1, \ldots, v_n\}$.*

Une interprétation I se représente souvent, de façon un peu abusive, par le sous-ensemble des variables propositionnelles v telles que $I(v) = t$ *i.e.*, par l'ensemble des variables propositionnelles vraies dans I, les autres variables apparaissant dans la formule étant considérées comme fausses dans I.

Définition 2.7 – Relation de satisfiabilité[4] – *Soit A une formule et I une interprétation de A, on définit la relation de satisfiabilité (notée $\models$) entre I et A par :*

- *si $A \in V_p$, $I \models A$ si et seulement si A appartient à I ;*
- *si A est de la forme $(\neg B)$, $I \models A$ ssi non[5] $(I \models B)$;*
- *si A est de la forme $(B \wedge C)$, $I \models A$ ssi $I \models B$ et $I \models C$;*
- *si A est de la forme $(B \vee C)$, $I \models A$ ssi $I \models B$ ou $I \models C$;*
- *si A est de la forme $(B \rightarrow C)$, $I \models A$ ssi (non $(I \models B)$) où $(I \models C)$;*
- *si A est de la forme $(B \leftrightarrow C)$, $I \models A$ ssi (non $(I \models B)$ et non $(I \models C)$) ou $(I \models B$ et $I \models C)$.*

Lorsque $(I \models A)$, on dit de façon équivalente que A est *satisfiable* par I, que I satisfait A, que I est un modèle de A ou que A est vraie dans I.

Cette définition correspond à une interprétation sémantique des différents connecteurs. Il devient possible d'associer à chaque connecteur une fonction à valeur dans $\{t, f\}$ qui permettra de calculer la valeur de vérité d'une formule construite via ce connecteur à partir de la valeur de vérité de ses sous-formules. C'est ce que montre la section qui suit.

2.4.3 Application pratique : les tables de vérité

Les tables de vérité des connecteurs dyadiques (s'appliquant à deux sous-formules) sont définies par la table 2.1. On peut lire ici, par exemple, que $(A \leftrightarrow B)$ est vrai lorsque A et B prennent les mêmes valeurs.

---

4 Il existe une querelle à l'intérieur de la communauté logique concernant la traduction du terme anglais *satisfiability*. Certains traduisent par *satisfaisabilité*, d'autres par *satisfiabilité*. Le même problème se pose pour l'adjectif *satisfaisable* ou *satisfiable*. Personnellement, nous préférons satisfaisable et satisfaisabilité, qui semblent plus proches des termes français *faisable, satisfaisant*. Mais le Larousse (édition 1988, page 909) ne reconnaît que les termes *satisfiable* et *satisfiabilité*. Nous nous sommes donc conformés à l'usage du dictionnaire.

5 non $(I \models A)$ signifie que l'on n'a pas $I \models A$: « non » appartient à la méta-langue.

A	$\neg A$
t	f
f	t

Table 2.2 – Négation ($\neg$)

Une interprétation qui a posé de nombreux problèmes est l'interprétation de l'implication, et plus particulièrement des troisième et quatrième lignes, qui nous dit que si A est faux alors $(A \rightarrow B)$ est vrai quel que soit B.

En termes communs, si l'on suppose que les propositions « les terriens sont tous verts », « $2 + 2 = 5$ » sont fausses et que la proposition « $2 + 2 = 4$ » est vraie alors la proposition « les terriens sont tous verts implique $2 + 2 = 5$ » est vraie, de même que « les terriens sont tous verts implique $2 + 2 = 4$ ».

L'acception courante du terme voudrait probablement que « si A alors B » soit vrai lorsque A et B sont vrais tous deux et soit faux lorsque A est faux. Il nous faudrait donc remettre en cause le t qui se trouve sur les lignes 3 et 4 de $(A \rightarrow B)$, et le remplacer par un f. Mais alors nous remarquons que $\rightarrow$ est devenu un synonyme de $\wedge$. Ce n'est donc pas de cette façon-là qu'il nous faudra résoudre ce « paradoxe » dit *paradoxe de l'implication matérielle*. Nous verrons plus loin que ce problème fut à l'origine d'une des premières logiques non-classiques formalisées.

Il faut d'autre part remarquer que dans bien des situations l'acception de « si ... alors ... » dans le langage courant recouvre bien la définition que nous donnons ici. Ainsi, soit la proposition : « s'il fait beau, j'irai me promener ». S'il fait effectivement beau, j'irai me promener et ma proposition sera vraie ; néanmoins, s'il pleut je n'irai pas me promener sans que toutefois la proposition que j'avais énoncée soit devenue un mensonge. L'implication matérielle (c'est ainsi qu'on la nomme) est donc bien reliée à une notion intuitive réelle. Nous nous attarderons davantage sur ce problème dans le chapitre consacré aux logiques non-classiques[6].

La table de vérité de $\neg$ est représentée dans la table 2.2.

Considérons par exemple la formule $F = p \rightarrow (q \vee (r \leftrightarrow (r \rightarrow \neg p)))$. La table de vérité de cette formule peut être calculée en considérant tous les n-uplets de valeurs de vérité possibles pour les variables qui apparaissent dans F. La table de vérité de F est ainsi construite dans la table 2.3.

2.4.4 Tautologies

Considérons la formule $p \rightarrow (q \rightarrow p)$ et construisons sa table de vérité, comme précédemment (cf. table 2.4).

On constate que la valeur de vérité de la formule est toujours t, indépendamment de la valeur assignée aux variables qui la composent. Une formule vérifiant cette propriété est appelée *tautologie*.

---

6 Remarquons déjà que pour construire une interprétation formelle de nos connaissances empiriques, nous sommes obligés de restreindre leur sens.

p	q	r	$\neg p$	$r \rightarrow \neg p$	$r \leftrightarrow (r \rightarrow \neg p)$	$q \vee (r \leftrightarrow (r \rightarrow \neg p))$	F
t	t	t	f	f	f	t	t
t	t	f	f	t	f	t	t
t	f	t	f	f	f	f	f
t	f	f	f	t	f	f	f
f	t	t	t	t	t	t	t
f	t	f	t	t	f	t	t
f	f	t	t	t	t	t	t
f	f	f	t	t	f	f	t

Table 2.3 – Table de vérité de : $F = p \rightarrow (q \vee (r \leftrightarrow (r \rightarrow \neg p)))$

p	q	$q \rightarrow p$	$p \rightarrow (q \rightarrow p)$
t	t	t	t
t	f	t	t
f	t	f	t
f	f	t	t

Table 2.4 – Table de vérité de : $p \rightarrow (q \rightarrow p)$

Définition 2.8 – Tautologie (formule valide) – *Une tautologie est une formule A vraie quelles que soient les valeurs de vérité des atomes qui la composent. On note alors $\models A$ (qui est en fait l'abréviation de $M \models A$ pour tout M).*

Les tautologies sont également appelées *formules valides*. On définit aussi :
- les *antitautologies* ou formules *insatisfiables* ou encore *sémantiquement inconsistantes* qui sont fausses dans toute interprétation ;
- les formules *satisfiables* ou *sémantiquement consistantes*[7] qui sont vraies dans au moins une interprétation ;
- les formules *contingentes* qui sont vraies dans certaines interprétations et fausses dans d'autres.

On notera que F est une tautologie si et seulement si $\neg F$ est inconsistante.

2.4.5 Substitution

Théorème 2.1 – Règle de substitution uniforme – *Soit la formule F contenant les atomes $p_1, p_2, \ldots, p_n$. Soit la formule F^* obtenue en substituant aux atomes $p_1, p_2, \ldots, p_n$ les formules $A_1, A_2, \ldots, A_n$. Alors si $\models F$, on a $\models F^*$.*

Ainsi, la formule $p \rightarrow (q \rightarrow p)$ étant valide, toute substitution de sous-formule à p ou q nous donnera une nouvelle formule valide. Nous pouvons donc aussi l'écrire : $A \rightarrow (B \rightarrow A)$ est valide quelles que soient les formules A et B.

---

7 Certains auteurs français emploient les termes *cohérence* et *cohérent* en lieu et place de *consistance* et *consistant*, qui proviennent directement de l'anglais *consistency*. Il est vrai que *cohérent* est probablement mieux adapté que *consistant*, mais l'usage ne semble pas avoir clairement imposé un des deux termes en français.

p	q	r	$(p \to q) \wedge (p \vee r)$	$(q \vee r)$	$(p \to r)$
t	t	t	t	t	t
t	t	f	t	t	f
t	f	t	f	t	t
t	f	f	f	f	f
f	t	t	t	t	t
f	t	f	f	t	t
f	f	t	t	t	t
f	f	f	f	f	t

Table 2.5 – Conséquence valide

Notons que la réciproque est bien entendu fausse. Ainsi $(p \wedge \neg p) \to q$ est valide, elle est de la forme $r \to s$ qui n'est pas une formule valide.

2.4.6 La notion de conséquence valide

Considérons la table de vérité 2.5 pour la formule $A = (p \to q) \wedge (p \vee r)$. Posons $B = (q \vee r)$ et $C = (p \to r)$.

On constate que si l'on nous donne l'information que A est vraie, alors, en consultant la table de vérité, nous savons que B est vraie. En revanche, cette même information ne permet de tirer aucune conclusion sur la valeur de vérité de C (il suffit de considérer les première et deuxième lignes de la table de vérité pour s'en convaincre). On dit alors que B est une conséquence valide de A, et l'on note $A \models B$[8]. De façon plus formelle :

Définition 2.9 – Conséquence valide – *Soit une formule B et une famille de n formules A_i ; soient $p_1, p_2, \ldots p_m$ les atomes figurant dans les A_i et dans B. Nous disons que B est une conséquence valide des A_i si dans les tables de vérité des A_i et de B la formule B a la valeur t sur toutes les lignes où les A_i sont simultanément à t. On note alors $A_1, A_2, \ldots, A_n \models B$.*

Théorème 2.2 – *$A \models B$ est équivalent à $\models (A \to B)$.*

Ce théorème permet de rattacher la notion de conséquence valide à la notion de tautologie.

Nous allons donner une démonstration rapide de ce théorème ; nous allons séparer la démonstration en deux parties :

- sur les lignes où A est f, $(A \to B)$ est t en fonction de la table de l'implication. Donc l'équivalence entre $A \models B$ et $\models (A \to B)$ est bien vérifiée sur ces lignes ;
- sur les lignes où A est t :
  - si $A \models B$, alors B est t sur ces lignes, et d'après la table de l'implication $(A \to B)$ est également vrai. Donc, si $A \models B$ alors $\models (A \to B)$;
  - réciproquement, si $\models (A \to B)$, alors $(A \to B)$ est t sur toutes les lignes où A est t ; donc, d'après la table de l'implication, B doit être t sur toutes les lignes où A est t. Donc $A \models B$.

8 La relation ainsi définie n'est pas une relation de satisfiabilité, même si les symboles employés sont les mêmes. Une relation de satisfiabilité relie interprétations et formules, la relation de conséquence relie des formules entre elles.

2.5 Théorie de la démonstration

La théorie de la démonstration est une théorie purement syntaxique. Par syntaxique, nous entendons que le mode de déduction utilisé est purement formel et se définit uniquement en termes d'application de règles mécaniques sur les formules (les éléments du langage), sans s'intéresser à leur valeur de vérité, à leur sens. Elle s'oppose en cela à la théorie des modèles, que nous venons de décrire, qui donne à chaque formule une valeur sémantique (c'est-à-dire une valeur de vérité, un sens).

Les règles utilisées sont appelées *règles d'inférence* (nous les retrouverons en logique des prédicats et pour le typage de termes en λ-calcul). Une règle d'inférence porte sur des *jugements* que l'on souhaite prouver mécaniquement, ici la véracité d'une formule.

Une règle d'inférence se présente sous la forme

Règle d'inférence 2.1 – *Nom de la règle*

$$\frac{j_1 \cdots j_n}{j}$$

où $j_1, \ldots, j_n$ sont des jugements qui forment les *prémisses* de la règle, j est un jugement qui forme sa conclusion. Une telle règle exprime que l'on peut produire (ou inférer) la conclusion de la règle si on a déjà pu produire ses prémisses. Dans le cas où $n = 0$, la règle est appelée un *axiome* et exprime que la conclusion peut être produite sans condition. La barre séparant prémisses et conclusion est alors omise.

Ainsi, si les jugements concernent des formules, il sera possible, à partir d'axiomes et de règles d'inférences, de produire de nouvelles formules que nous appellerons théorèmes, qui pourront eux aussi servir à produire de nouveaux théorèmes... Les axiomes sont les principes fondamentaux de la logique, et les autres règles d'inférence assurent la partie créative du mécanisme déductif.

Il n'y a derrière ce mécanisme aucune notion de vérité ou de fausseté des énoncés. Nous pouvons même considérer les symboles de connecteurs comme purement formels : dans le modèle formel, le symbole d'*équivalence* pourrait être remplacé par *Arthur*.

Bien entendu, une théorie axiomatique est construite pour être « en bonne correspondance » avec une théorie sémantique. Nous étudierons dans la section 2.6 ce que l'on entend par « bonne correspondance » entre une théorie axiomatique formelle et une interprétation sémantique.

2.5.1 Axiomatique du calcul propositionnel

Définition 2.10 – Théorème – *Une formule A est un théorème, noté $\vdash A$, si A est un axiome ou si A est obtenue à partir de l'application d'une règle d'inférence sur d'autres théorèmes.*

Nous allons maintenant introduire les axiomes[9] du calcul propositionnel :

9 Dans la présentation axiomatique que nous utilisons ici, il vaudrait mieux parler de *schéma d'axiome*. En effet, nous utilisons des lettres majuscules dans l'énoncé de ces axiomes, ce qui signifie que nous pouvons substituer

Axiome 2.1 –

$$A \rightarrow (B \rightarrow A)$$

Axiome 2.2 –

$$(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$$

Axiome 2.3 –

$$(\neg B \rightarrow \neg A) \rightarrow (A \rightarrow B)$$

Cet ensemble d'axiomes est l'ensemble de cardinal minimal[10] que l'on puisse adopter. On remarque qu'un certain nombre de connecteurs ($\leftrightarrow$,$\wedge$,$\vee$) n'apparaissent pas dans ces axiomes. En fait, ces connecteurs sont explicitement définis comme des synonymes d'autres expressions. Ainsi $(A \wedge B)$, $(A \vee B)$ et $(A \leftrightarrow B)$ sont définis respectivement par $\neg(A \rightarrow \neg B)$, $(\neg A \rightarrow B)$ et $\neg((A \rightarrow B) \rightarrow \neg(B \rightarrow A))$.

Nous devons également rappeler que A, B et C, dans les schémas d'axiomes précédents, représentent n'importe quelle formule de notre langage. Nous pouvons donc substituer à ces éléments formels n'importe quelle formule de notre langage formel *i.e.*, nous pouvons substituer n'importe quoi à A, B et C ; la formule résultante sera un théorème de notre logique. Ainsi considérons $A \rightarrow (B \rightarrow A)$; nous pouvons remplacer A par $(p \rightarrow q)$ et B par $(r \rightarrow s)$. Nous obtenons alors : $(p \rightarrow q) \rightarrow ((r \rightarrow s) \rightarrow (p \rightarrow q))$ qui est un théorème.

Nous allons maintenant nous intéresser à la partie déductive du langage ;

Règle d'inférence 2.2 – *Règle du modus-ponens*

$$\frac{\vdash A, \vdash A \rightarrow B}{\vdash B}$$

La règle du modus-ponens peut se lire : « si A est un théorème et si $A \rightarrow B$ est un théorème alors B est un théorème ». On voit ici que la règle d'inférence permet de construire de nouveaux théorèmes, comme annoncé.

2.5.2 Démonstrations

Définition 2.11 – Démonstration – *Une démonstration d'un théorème A est une suite finie $(A_1, A_2, \ldots A_i, \ldots A)$ où chaque A_i est soit un axiome, soit obtenu par l'application*

---

n'importe quelle formule à la lettre en question. Il ne s'agit donc pas à proprement parler d'un axiome (qui serait par exemple : $(p \rightarrow (q \rightarrow r))$) mais bien d'un schéma général. Ainsi la règle de substitution uniforme est implicitement présente. On peut utiliser une autre présentation en donnant de véritables axiomes et en ajoutant un principe général de substitution.

10 On peut se demander ce que veut dire *minimal*. Nous devons nous rappeler que nous construisons ce système formel afin qu'il soit adéquat à l'interprétation usuelle que nous avons du calcul des propositions. Le jeu d'axiomes ci-dessus est le jeu de plus petit cardinal connu qui atteigne ce but.

d'une règle d'inférence sur des théorèmes A_j et A_k précédemment produits $(j, k < i)$. On note alors $\vdash A$ (A est un théorème).

Cette notion de démonstration correspond bien à l'idée intuitive que nous nous faisons d'une démonstration : un enchaînement fini d'inférences formelles.

Nous allons démontrer formellement sur un exemple : $\vdash (A \rightarrow A)$.

1. $\vdash ((A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C)))$: axiome 2.2
2. $\vdash ((A \rightarrow ((A \rightarrow A) \rightarrow A)) \rightarrow ((A \rightarrow (A \rightarrow A)) \rightarrow (A \rightarrow A)))$: en substituant dans (1) $(A \rightarrow A)$ à B et A à C.
3. $\vdash A \rightarrow (B \rightarrow A)$: axiome 2.1
4. $\vdash A \rightarrow ((A \rightarrow A) \rightarrow A)$: en substituant $(A \rightarrow A)$ à B dans (3).
5. $\vdash ((A \rightarrow (A \rightarrow A)) \rightarrow (A \rightarrow A))$: en appliquant la règle d'inférence entre (4) et (2).
6. $\vdash A \rightarrow (A \rightarrow A)$: axiome 2.1 où l'on substitue A à B.
7. $\vdash A \rightarrow A$ en appliquant la règle d'inférence entre (5) et (6).

On constate qu'une démonstration formelle est techniquement relativement fastidieuse.

2.5.3 Déduction

La notion de déduction résulte d'un prolongement de la notion de démonstration :

Définition 2.12 – Déduction – *On dit qu'une formule A se déduit d'un ensemble de formules $(B_1, B_2, \ldots, B_i, \ldots, B_n)$, et l'on note : $B_1, B_2 \ldots B_n \vdash A$ s'il existe une suite finie $(A_1, A_2, \ldots A_i, \ldots A)$ où chaque A_i est soit un axiome, soit un des B_i, soit obtenu par l'application d'une règle d'inférence*[11] *sur deux éléments A_j, A_k de la suite déjà obtenue $(j, k < i)$.*

Les B_i sont fréquemment appelés *hypothèses*. La notion de démonstration et la notion de déduction sont donc très proches, la principale différence étant dans le nombre d'hypothèses que nous utilisons. Si nous n'avons aucune hypothèse, il s'agit d'une démonstration, si nous avons une hypothèse au moins, il s'agira d'une déduction.

Théorème 2.3 – Théorème de déduction – *Si $A \vdash B$ alors $\vdash (A \rightarrow B)$*

Ce théorème est également connu sous le nom de théorème d'Herbrand[12], car il fut établi pour la première fois par Herbrand en 1930. Nous l'admettrons.

2.6 Propriétés fondamentales

Nous allons dans cette section nous efforcer principalement de montrer que la théorie des modèles et la théorie de la démonstration sont équivalentes en calcul propositionnel. En effet, il est essentiel de vérifier que l'on peut indifféremment utiliser l'une ou l'autre méthode pour prouver la « vérité » d'une formule. Cette « équivalence » ou « bonne correspondance » repose sur un certain nombre de propriétés que possède la logique propositionnelle.

---

11 Dans le cas présent (logique classique propositionnelle), il n'y a qu'une règle d'inférence. Ce n'est pas toujours le cas, comme nous le verrons.

12 Notons qu'il existe de nombreux théorèmes d'Herbrand en logique mathématique et en théorie de la résolution.

2.6.1 Adéquation

Étant donnés un langage et une théorie des modèles (sémantique), on peut s'intéresser à certaines qualités d'une théorie de la preuve.

Définition 2.13 – Adéquation – *Une logique est dite adéquate*[13] *si tout théorème* A ($\vdash A$) *est une formule valide* ($\models A$).

Théorème 2.4 – Adéquation du calcul propositionnel – *Le calcul propositionnel est adéquat* i.e., *si* $\vdash A$ *alors* $\models A$.

En effet, par induction :
- les axiomes sont valides ;
- si A est obtenu par *modus ponens* à partir de de $\vdash B$ et de $\vdash (B \rightarrow A)$, alors (par hypothèse d'induction) $\models B$ et $\models (B \rightarrow A)$. On en déduit que $\models A$.

Notons que la notion d'adéquation est *toujours relative à une théorie des modèles.*

2.6.2 Consistance

Définition 2.14 – Consistance – *Une logique est dite (syntaxiquement) consistante s'il n'existe aucune formule* A *du langage telle que* $\vdash A$ *et* $\vdash \neg A$.

Cette propriété est une conséquence directe de la propriété d'adéquation démontrée au-dessus. En effet, supposons que nous ayons simultanément $\vdash A$ et $\vdash \neg A$. Alors d'après le théorème 2.4 nous avons $\models A$ et $\models \neg A$, ce qui nous impose d'avoir dans la table de vérité de A et de $\neg A$ des t sur chaque ligne, ce qui est impossible.

Théorème 2.5 – Consistance du calcul propositionnel – *Le calcul des propositions est consistant.*

La consistance est une propriété purement formelle qui peut se démontrer par des moyens syntaxiques (même si nous avons ici utilisé l'adéquation pour la démontrer plus simplement).

La notion de consistance syntaxique s'étend aux ensembles de formules : on dira qu'un ensemble de formules E est syntaxiquement consistant dans une logique donnée s'il n'existe pas de formule A telle que $E \vdash A$ et $E \vdash \neg A$. Le théorème 2.4 d'adéquation peut alors se reformuler, de façon équivalente en « tout ensemble de formules satisfiable est consistant ».

2.6.3 Complétudes

Définition 2.15 – Complétude forte – *Une logique est dite fortement complète ssi la relation de conséquence valide correspond à la relation de dérivabilité : Soit* E *un ensemble de formules, si* $E \models A$ *alors* $E \vdash A$.

Définition 2.16 – Complétude faible – *Une logique est dite faiblement complète si toute formule valide est un théorème* i.e., *quand* $\models A$ *alors* $\vdash A$.

---

13 Les Anglais emploient le terme de *sound* (adéquat) et *soundness* (adéquation). Le premier est parfois traduit en français par le mot « *sain* ».

La complétude faible d'une logique garantit que les règles de raisonnement formalisées par cette logique suffisent pour démontrer toutes les propriétés valides (au sens de la théorie des modèles utilisée) exprimables dans son langage. La complétude forte implique bien sûr la complétude faible.

Théorème 2.6 – *Le calcul propositionnel est fortement complet.*

Nous ne démontrerons pas ce théorème. La démonstration complète se trouve (par exemple) dans (Kleene 1987). En terme de consistance syntaxique, le théorème précédent peut se réécrire de façon équivalente en « tout ensemble de formules consistant est satisfiable ».

Il ne faut pas confondre cette notion de complétude (liant syntaxe et sémantique) avec la notion de complétude *syntaxique* (ou complétude d'une théorie) que nous utiliserons dans le chapitre 5 et définie par :

Définition 2.17 – Complétude syntaxique – *Une logique est dite* syntaxiquement complète[14] *ssi, pour toute formule A, on a soit* $\vdash A$, *soit* $\vdash \neg A$.

Le calcul propositionnel n'est pas syntaxiquement complet, puisqu'il existe des formules F contingentes (ni tautologiques, ni antitautologiques : un littéral par exemple) pour lesquelles on a ni $\vdash F$, ni $\vdash \neg F$ (du fait de la complétude).

2.6.4 Décidabilité

Nous introduisons rapidement et informellement la notion de décidabilité, que nous retrouverons plus formellement dans le chapitre 5.

Définition 2.18 – *On dit qu'une logique est décidable quand on connaît une procédure mécanique permettant d'établir en un temps* fini *si une formule donnée est ou n'est pas un théorème.*

Le calcul propositionnel est décidable, car une formule est un théorème si et seulement si elle est valide, et la validité d'une formule peut être établie mécaniquement en un temps fini par la méthode des tables de vérité : en effet, le nombre de modèles à tester est fini puisque le nombre de variables propositionnelles intervenant dans une formule est fini.

2.6.5 Remarques

Les propriétés d'adéquation et de complétude permettent de relier l'interprétation sémantique (théorie des modèles) et l'interprétation syntaxique (théorie de la démonstration) de la logique. En revanche, la propriété de consistance est purement formelle et ne se rapporte qu'à la théorie de la démonstration[15].

Nous avons ici établi que $\vdash B$ était équivalent à $\models B$. Ceci justifie le fait (que nous ressentions déjà intuitivement) qu'il est équivalent de démontrer une formule du calcul

---

14 Il faut noter que toute logique complète est décidable (cf. *infra*) : il suffit de chercher en parallèle une preuve de A et de $\neg A$.

15 Notons que chez certains auteurs, ce que nous appelons adéquation est appelée consistance, et ce que nous appelons consistance est appelée consistance simple.

Théorie de la démonstration	Théorie des modèles
Interprétation syntaxique	*Interprétation sémantique*
Axiomes, règles d'inférence	*Table de vérité définissant les connecteurs*
Formule	*Table de vérité de la formule*
Théorème	*Tautologie*
Déduction	*Conséquence valide*

Table 2.6 – Liens entre syntaxe et sémantique

propositionnel ou de calculer sa table de vérité. De façon plus philosophique, cela signifie que les notions de véracité et de démontrabilité sont équivalentes. Cela nous paraît évident : si une proposition est valide, on doit pouvoir la démontrer et réciproquement, toute proposition que l'on parvient à démontrer doit être valide. C'est même dans ce but que nous avons construit notre modèle syntaxique. Le tableau 2.6 résume de façon grossière les différentes « correspondances ».

Des générations de mathématiciens ont d'ailleurs pensé qu'il était toujours possible de construire un système axiomatique adapté à une théorie des modèles pourvu que celle-ci soit « raisonnable ». Nous verrons cependant que dans des cas plus complexes que le calcul des propositions, les notions de démontrabilité et de vérité ne peuvent se recouvrir totalement.

CHAPITRE 3

Le calcul des prédicats

Jean-Marc Alliot — Thomas Schiex

3.1 Introduction

Dans le chapitre précédent, nous nous sommes intéressés aux propositions comme éléments de base de notre langage. Nous allons dans ce chapitre enrichir notre analyse de ces propositions en introduisant une nouvelle structure, celle de prédicat. Un prédicat formalise une notion appelée en grammaire *structure sujet-prédicat.*

Ainsi, considérons la proposition : « Les schtroumpfs sont bleus ». La structure « sont bleus » est un prédicat et « schtroumpfs » est le sujet du prédicat. On peut aussi écrire « x est bleu » auquel cas notre prédicat apparaît comme une *fonction propositionnelle* à une variable, fonction qui prend la valeur t (vraie) lorsque x est un schtroumpf et f (faux) lorsque x est un homme. Nous parlerons à l'avenir plutôt de prédicat que de fonction propositionnelle.

Le calcul des prédicats permet de s'intéresser à des énoncés relatifs à un ensemble d'objets, comme « tous les schtroumpfs sont bleus » ou à certains éléments de ces ensembles, comme « il existe des schtroumpfs gourmands ». Les notions de « quel que soit » et de « il existe » sont fondamentales en calcul des prédicats.

L'ensemble des théorèmes du calcul des prédicats est un sur-ensemble de l'ensemble des théorèmes du calcul propositionnel ; en particulier l'ensemble des théorèmes démontrés sur les propositions dans le chapitre précédent restera valable, une proposition étant un prédicat d'un type particulier, sans variable. De même, la base axiomatique du calcul des prédicats sera celle du calcul des propositions auquel nous nous contenterons d'adjoindre de nouveaux axiomes.

Nous allons, dans ce chapitre, étudier d'abord la structure formelle du langage (formules bien formées), puis la théorie de la démonstration en calcul des prédicats, la théorie des modèles et nous énoncerons enfin les résultats concernant la consistance et la complétude.

Cependant, nous ne développerons pas les démonstrations des théorèmes, et notre exposé sera beaucoup plus succinct et plus informel qu'il ne l'était pour le calcul propositionnel, le calcul des prédicats étant seulement à nos yeux un point de passage obligé avant de définir les systèmes formels et de présenter la plupart des résultats de la programmation logique. Un certain nombre de propriétés et de définitions seront donc énoncées afin de fournir une base suffisante à la suite de cet ouvrage.

Notons enfin que le calcul des prédicats est aussi appelé *logique du premier ordre.*

3.2 Définitions

Alors que le calcul propositionnel s'appuyait seulement sur des atomes et des connecteurs, le calcul des prédicats s'appuie sur six classes de symboles :

- **les variables :** nous les noterons généralement x, y, z...
- **les symboles de constantes :** nous les noterons a, b, c,...
- **les symboles de fonctions :** les symboles de fonctions d'arité $n > 0$ (ou encore à n places) seront notés f, g...
- **les symboles de prédicats :** les symboles de prédicats, d'arité positive ou nulle[1], seront notés p, q...
- **les connecteurs :** les connecteurs sont ceux du calcul propositionnel ;
- **les quantificateurs :** les quantificateurs (ou quanteurs) sont $\forall$ et $\exists$.

Nous définissons maintenant la notion de terme :

Définition 3.1 – Terme – *Un terme est défini récursivement par :*

- *une variable est un terme.*
- *un symbole de constante est un terme.*
- *si f est un symbole de fonction d'arité n et si t_1, t_2,...,t_n sont des termes, alors $f(t_1, t_2, \ldots, t_n)$ est un terme.*

Tout terme est obtenu par l'application des règles précédentes un nombre fini de fois.

La définition des formules[2] en calcul des prédicats est très proche de la définition des formules en calcul propositionnel. Nous commençons par définir les atomes :

Définition 3.2 – Atome – *Si p est un symbole de prédicat d'arité n et que t_1, t_2,...,t_n sont des termes, alors $p(t_1, t_2, \ldots, t_n)$ est un atome, ou formule atomique.*

Définition 3.3 – Formule – *Une formule est définie récursivement par :*

- *toute formule atomique est une formule ;*
- *si A et B sont des formules alors $(A \rightarrow B)$ est une formule, $(A \leftrightarrow B)$, $(A \wedge B)$ et $(A \vee B)$ sont des formules ;*
- *si A est une formule alors $(\neg A)$ est une formule ;*
- *si A est une formule, et si x est une variable quelconque alors $(\forall x\ A)$ est une formule. On dit que A est la* portée *(ou scope[3]) du quantificateur $\forall x$;*
- *si A est une formule, et si x est une variable quelconque alors $(\exists x\ A)$ est une formule. On dit que A est la* portée *du quantificateur $\exists x$.*

Toute formule est obtenue par l'application des règles précédentes un nombre fini de fois.

Les mêmes remarques que celles faites au chapitre précédent concernant l'ordre de dominance s'appliquent. Les symboles $\forall$ et $\exists$ viennent en dernier dans la liste. Ainsi $\forall x\ A \vee B$ doit se lire $(\forall x\ A) \vee B$.

3.2.1 Variables liées, variables libres

Considérons la formule élémentaire $\forall x\, p(x, y)$ où p est un symbole de prédicat à deux variables.

---

1 Les prédicats d'arité nulle sont équivalents aux atomes du calcul propositionnel.

2 ou formules bien formées

3 *scope* est un mot anglais que l'on trouve fréquemment employé, même en français.

L'unique occurrence de la variable x dans $p(x, y)$ est placée dans la *portée* du quantificateur $\forall$. Une telle occurrence de variable est dite *liée.*

Définition 3.4 – Variables liées – *L'ensemble des variables liées d'une formule A (noté $\mathcal{B}(A)$) se définit de façon inductive par :*

- *si A est un atome, $\mathcal{B}(A) = \varnothing$;*
- *si A est de la forme $(B \to C)$, $(B \leftrightarrow C)$, $(B \wedge C)$ ou $(B \vee C)$, alors $\mathcal{B}(A) = \mathcal{B}(B) \cup \mathcal{B}(C)$;*
- *si A est de la forme $(\neg B)$, alors $\mathcal{B}(A) = \mathcal{B}(B)$;*
- *si A est de la forme $(\forall x B)$ ou $(\exists x B)$, alors $\mathcal{B}(A) = \{x\} \cup \mathcal{B}(B)$.*

On définit de la même façon les variables libres d'une formule par :

Définition 3.5 – Variables libres – *L'ensemble des variables libres d'une formule A (noté $\mathcal{F}(A)$) se définit de façon inductive par :*

- *si A est un atome, $\mathcal{F}(A)$ est égal à l'ensemble des variables apparaissant dans A ;*
- *si A est de la forme $(B \to C)$, $(B \leftrightarrow C)$, $(B \wedge C)$ ou $(B \vee C)$, alors $\mathcal{F}(A) = \mathcal{F}(B) \cup \mathcal{F}(C)$;*
- *si A est de la forme $(\neg B)$, alors $\mathcal{F}(A) = \mathcal{F}(B)$;*
- *si A est de la forme $(\forall x B)$ ou $(\exists x B)$, alors $\mathcal{F}(A) = \mathcal{F}(B) - \{x\}$.*

Par exemple, si l'on considère la formule $A = p(x, y) \vee \forall x\,(q(x) \wedge r(x))$, on a $\mathcal{B}(A) = \{x\}$ et $\mathcal{F}(A) = \{x, y\}$. Une variable peut donc être à la fois libre et liée.

Définition 3.6 – Formule close – *Une formule A telle que $\mathcal{F}(B) = \varnothing$ est dite close, fermée ou est encore appelée* énoncé.

3.2.2 Clôture d'une formule

Nous allons rencontrer rapidement une difficulté concernant les variables libres. Il est en effet possible de « comprendre » une variable libre de deux façons dans une formule. Cette différence est du même ordre que la différence en mathématique entre *identité* et *égalité conditionnelle.*

Ainsi la formule $(x + 1)^2 = x^2 + 2x + 1$ est une identité alors que la formule $x^2 - 4x + 4 = 0$ est une égalité conditionnelle. Dans la première formule nous dirons que la variable x est prise au *sens de la généralité*, alors que dans la seconde nous dirons qu'elle est prise *au sens conditionnel.*

Ainsi, en logique des prédicats, si un théorème est démontré à partir d'une hypothèse $A(x)$ où x est pris dans le sens conditionnel, c'est à dire où x est contraint à prendre ses valeurs dans un sous-ensemble des valeurs possibles, alors le théorème ne s'applique que pour le même x. En revanche un théorème démontré à partir de $A(x)$ où x est pris au sens de la généralité est valable pour tout x. Il faut bien prendre garde à ce type de phénomènes, source de sophismes logiques. Il est clair que nombre de problèmes proviennent de l'emploi de variables libres. Si nous nous contraignions à n'utiliser que des variables liées l'ambiguïté disparaîtrait sur le champ[4].

On peut toujours ramener une formule non close à une formule close via les deux notions suivantes :

4 Ce n'est malheureusement pas toujours possible dans les démonstrations...

Définition 3.7 – Clôture universelle – *On appelle clôture universelle de la formule A la formule liée $\forall x_1 \forall x_2 \ldots \forall x_q\, A$ où $x_1,x_2,\ldots,x_q$ sont les variables libres apparaissant dans A. On note parfois la clôture universelle de A : $\forall A$.*

On parle également de clôture existentielle :

Définition 3.8 – Clôture existentielle – *On appelle clôture existentielle de la formule A la formule liée $\exists x_1 \exists x_2 \ldots \exists x_q\, A$ où $x_1,x_2,\ldots,x_q$ sont les variables libres apparaissant dans A. On note parfois la clôture existentielle de A : $\exists A$.*

3.3 Théorie de la démonstration

La théorie de la démonstration du calcul des prédicats découle de la théorie de la démonstration en calcul propositionnel.

3.3.1 Axiomatique du calcul des prédicats

Nous emploierons, dans la suite, la notation suivante : $A(x)$ désigne une formule contenant x comme variable libre ($x \in \mathcal{F}(A)$). Dans ce cadre, $A(t)$ désigne toute formule obtenue en substituant le terme t à toutes les occurrences de x dans la formule $A''(x)$ obtenue à partir de $A(x)$:

1. en renommant les occurrences liées de x dans leur portée de telle manière que x ne soit plus liée. On obtient ainsi $A'(x)$;
2. en renommant dans $A'(x)$ les variables apparaissant dans le terme t et liées dans $A'(x)$ (afin qu'il n'existe plus de variables de t qui soient liées). On obtient ainsi $A''(x)$.

Les notions, encore peu formelles, de substitution et de renommage de variables liées seront formalisées dans le chapitre 7.

Si $x \notin \mathcal{F}(A)$, on convient que $A = A(x) = A(t)$.

Exemple : si $A(x) = (q(x) \rightarrow \forall z \exists x\, p(x,z))$ et $t = f(z,y)$.

1. x étant liée dans la sous-formule $\exists x\, p(x,z')$, elle est renommée en x'. On obtient $A'(x) = (q(x) \rightarrow \forall z \exists x'\, p(x',z)$.
2. il faut renommer les variables apparaissant dans t liées dans $A(x)$. Il s'agit de z qui sera renommée en z'. On obtient la formule $A''(x) = (q(x) \rightarrow \forall z' \exists x'\, p(x',z'))$.

Il suffit alors d'appliquer la substitution de t à x pour obtenir $A(t) = (q(f(z,y)) \rightarrow \forall z' \exists x'\, p(x',z')$.

Les schémas d'axiomes de la logique des prédicats sont alors obtenus en ajoutant aux schémas d'axiomes du calcul propositionnel les schémas d'axiomes suivants :

Axiome 3.1 – Schéma 4 –

$$\forall x\, A(x) \rightarrow A(t)$$

Axiome 3.2 – Schéma 5 –

$$((A \rightarrow B) \rightarrow (A \rightarrow \forall x\, B))$$

où $x \notin \mathcal{F}(A)$.

Nous devons également ajouter au *modus ponens* une règle d'inférence appelée règle de généralisation :

Règle d'inférence 3.1 – Généralisation –

$$\frac{\vdash A}{\vdash \forall x\, A}$$

3.3.2 Démonstration – Déduction

Les définitions des démonstrations, des déductions sont les mêmes[5] que dans le cadre de la logique des propositions (elles peuvent, en fait, être données dans le cadre général de « systèmes de déduction » ou « systèmes formels » dont la logique propositionnelle et la logique des prédicats sont deux instances).

3.3.3 Théorème de la déduction

Ce théorème déjà énoncé en logique propositionnelle s'étend à la logique des prédicats. Une restriction est nécessaire pour traiter un problème introduit par les variables libres :

Théorème 3.1 – Théorème de déduction – *Soit* A *une formule close. Si* $A \vdash B$ *alors* $\vdash A \rightarrow B$.

3.4 Théorie des modèles

3.4.1 Introduction informelle

Comme en calcul propositionnel, nous allons nous efforcer de construire un modèle permettant de dégager une interprétation sémantique de nos formules.

En calcul des prédicats, toutefois, il n'est pas possible d'appliquer une méthode des tables de vérité directement dérivée du calcul propositionnel, en raison des domaines de valeur des variables de chacun des prédicats. En fait, pour une formule atomique $p(x_1, \ldots, x_n)$, nous allons avoir besoin d'une fonction (appelée fonction d'interprétation) chargée de donner un sens au symbole p, et donc de calculer sa valeur de vérité selon la valeur des $x_1, \ldots, x_n$.

Ainsi, soit le prédicat p « est bleu » opérant sur le domaine {martien, homme, schtroumpf}. Une interprétation *possible* (notée $I(p)$) de p est donnée dans la table 3.1.

---

5 Il y a plusieurs façon, plus ou moins élégantes, de définir la déduction en logique des prédicats et à chacune correspond un ensemble d'autres modifications légères (satisfiabilité, restriction sur la généralisation,...) visant toutes à conserver les bonnes correspondances entre syntaxe et sémantique. Ces variantes proviennent en fait des différentes interprétations possibles d'une formule non fermée et de la règle de généralisation. Ces définitions sont heureusement équivalentes du point de vue de la déduction quand les hypothèses sont des formules closes, ou plus généralement quand aucune généralisation n'est effectuée sur une variable libre d'une hypothèse. De même, les ensembles de théorèmes (et donc de formules valides) correspondent, que les formules soient libres ou non. Le lecteur pourra comparer, par exemple, la présentation de (Delahaye 1986) avec celle de (Enderton 1972).

x	$I(p)(x)$
martien	f
homme	f
schtroumpf	t

Table 3.1 – Une interprétation de p

3.4.2 Interprétation

La notion d'interprétation en calcul des prédicats est proche de celle en calcul propositionnel :

Définition 3.9 – Interprétation – *Une interprétation d'une formule F est caractérisée par la donnée d'un ensemble de définition $\mathcal{D}$ non vide, appelé le domaine de l'interprétation et d'une fonction d'interprétation I qui associe :*
- *à chaque symbole de constante de F, un élément de $\mathcal{D}$;*
- *à chaque symbole de fonction f d'arité n de F, le graphe d'une fonction de $\mathcal{D}^n \rightarrow \mathcal{D}$ définissant f ;*
- *à chaque symbole de prédicat p d'arité n de F, le graphe d'une fonction de $\mathcal{D}^n \rightarrow \{t, f\}$ définissant p.*

Définition 3.10 – Interprétation des termes – *Soit une formule F et I une interprétation de cette formule. On peut étendre l'interprétation I aux termes de F :*
- *à chaque symbole de constante, on associe sa valeur selon I ;*
- *à chaque variable, on associe la variable elle-même ;*
- *à chaque terme $f(t_1, \ldots, t_n)$, on associe le terme $f'(t'_1, \ldots, t'_n)$ où $t'_1, \ldots, t'_n$ sont les interprétations des $t_1, \ldots, t_n$ et f' est l'interprétation de f.*

Cette définition étant posée, il est maintenant possible d'utiliser la méthode des tables de vérité sur les formules. Un problème est posé par les variables libres. On étend l'interprétation aux formules F_i ayant i variables libres. $I(F_i)$ est alors une application dont les arguments sont les valeurs des i variables libres.

Définition 3.11 – Interprétation des formules – *Elle est définie par :*
- *si F est un atome $p(t_1, \ldots, t_n)$, $I(F)$ est la fonction $p'(t'_1, \ldots, t'_n)$ où p' est l'interprétation de p et où chaque t'_i est l'interprétation de t_i ;*
- *si F est de la forme $(\neg G)$, $(G \rightarrow H)$, $(G \leftrightarrow H)$, $(G \wedge H)$ et $(G \vee H)$, $I(F)$ est définie par les mêmes lois fonctionnelles que celles définies pour le calcul propositionnel ;*
- *si F est de la forme $\forall x\, G(x, y_1, \ldots, y_n)$, pour tout n-uplet $(a_1, \ldots, a_n) \in \mathcal{D}^i$, $I(F)(a_1, \ldots, a_n) = t$ si pour toute valeur $a \in \mathcal{D}$, $I(G)(a, a_1, \ldots, a_n) = t$. $I(F)(a_1, \ldots, a_n) = f$ sinon ;*
- *si F est de la forme $\exists x\, G(x, y_1, \ldots, y_n)$, pour tout n-uplet $(a_1, \ldots, a_n) \in \mathcal{D}^i$, $I(F)(a_1, \ldots, a_n) = t$ s'il existe une valeur $a \in \mathcal{D}$, $I(G)(a, a_1, \ldots, a_n) = t$. $I(F)(a_1, \ldots, a_n) = f$ sinon.*

Pour une interprétation donnée, à toute formule close se trouve associée une des 2 fonctions sans arguments (des constantes) t ou f. Pour une formule possédant des variables libres et dont l'interprétation est une fonction constante égale à t, on écrira $I(F) = t$.

x	$I_1(p)(x)$	$I_2(p)(x)$	$I_3(p)(x)$	$I_4(p)(x)$
1	t	t	f	f
2	t	f	t	f

Table 3.2 – Interprétations possibles

3.4.3 Formule valide

La définition de validité d'une formule est similaire à celle introduite en calcul propositionnel :

Définition 3.12 – Formule valide – *Une formule F est dite valide ssi, pour toute interprétation I, on a $I(F) = t$. On note alors $\models F$.*

Le calcul de l'interprétation (étant donnés un domaine et la fonction d'interprétation) d'une formule peut donc se faire via une table de vérité, comme en calcul des propositions. Cependant, la détermination de la validité est rendue délicate par le nombre possible de domaines et la taille de ceux-ci.

3.4.4 Exemple de table de vérité

Soit la formule $A = p(y) \vee \forall x\,(p(x) \rightarrow q)$ où p est défini sur le domaine $\mathcal{D} = \{1, 2\}$. Nous allons essayer de montrer que cette formule n'est pas valide. Pour cela nous allons construire la table de vérité de A. Tout d'abord nous devons considérer les quatre interprétations possibles pour p (table 3.2). Nous pouvons ensuite nous lancer dans la construction de la table de vérité de A (table 3.3) en prenant en compte les différentes interprétations possibles de p et de q et les différentes valeurs possibles de y.

Détaillons le calcul des deux premières lignes par exemple. L'interprétation de p utilisée est I_1 et on suppose que l'interprétation de q est t. L'interprétation de $p(x) \rightarrow q$ est une fonction d'une variable (la valeur de x) toujours égale à t d'après l'interprétation I_1, on en déduit que la formule close $\forall x\,(p(x) \rightarrow q)$ s'interprète par la fonction constante t. L'interprétation de $p(y)$ est une fonction d'une variable (la valeur de y) constante égale à t (d'après I_1). On en déduit que sous I_1, l'interprétation de la formule A est la fonction d'une variable constante et égale à t. D'où les deux premières lignes.

On déduit du contenu de la table que la formule n'est pas valide puisqu'il existe un domaine et une interprétation I (telle que $I(p) = I_2(p)$ et $I(q) = f$) pour lesquels l'interprétation de A n'est pas la fonction constante t.

3.4.5 Satisfiabilité, insatisfiabilité et conséquence

Les notions introduites en logique des propositions s'étendent en logique des prédicats. Encore une fois, les formules non closes sont sources de quelques problèmes.

Définition 3.13 – Formule satisfiable – *Une formule A sera dite satisfiable, ou sémantiquement consistante, s'il existe une interprétation I telle que $I(A) = t$. L'interprétation est alors un modèle de A.*

Cette notion est étendue aux ensembles de formules. Un ensemble de formules est dit consistant ssi il existe *un* modèle de toutes les formules de l'ensemble.

$I(p)$	$I(q)$	y	$I(p(y) \vee \forall x\,(p(x) \rightarrow q))(y)$
$I_1(p)$	t	1	t
$I_1(p)$	t	2	t
$I_1(p)$	f	1	t
$I_1(p)$	f	2	t
$I_2(p)$	t	1	t
$I_2(p)$	t	2	t
$I_2(p)$	f	1	t
$I_2(p)$	f	2	f
$I_3(p)$	t	1	t
$I_3(p)$	t	2	t
$I_3(p)$	f	1	f
$I_3(p)$	f	2	t
$I_4(p)$	t	1	t
$I_4(p)$	t	2	t
$I_4(p)$	f	1	t
$I_4(p)$	f	2	t

Table 3.3 – Table de vérité de $p(y) \vee \forall x\,(p(x) \rightarrow q)$

De la même façon, une formule est dite *insatisfiable* ou *sémantiquement inconsistante* ssi elle ne possède pas de modèle. Une formule *close* A est valide ssi sa négation est *insatisfiable.*

Définition 3.14 – Conséquence logique – *Soit une formule B et une famille de n formules A_i; On dit que B est conséquence des A_i si pour toute interprétation I telle que $\forall A_i, I(A_i) = t$, on a aussi $I(B) = t$. On note alors $A_1, A_2, \ldots, A_n \models B$.*

Autrement dit, tout modèle des A_i est un modèle de B.

3.4.6 Le problème des domaines infinis

Vérifier la validité d'une formule dans le cas où les domaines de ses variables sont finis ne pose aucun problème, du moins d'ordre théorique[6]. Il suffit de construire, comme en calcul propositionnel, une table de vérité.

En revanche si un ou plusieurs des domaines sont infinis, le problème est tout autre. On ne peut plus construire de table de vérité et l'on est forcé de tenir des raisonnements d'ordre général pour démontrer la validité d'une formule (pour démontrer qu'elle n'est pas valide, il suffit bien sûr d'exhiber un cas où elle est fausse). On parle alors d'évaluation de la table de vérité plutôt que de calcul[7].

Cette particularité rend la théorie des modèles du calcul des prédicats beaucoup plus délicate à manipuler que la théorie des modèles du calcul propositionnel.

---

6 Nous reviendrons sur les problèmes pratiques dans les chapitres suivants, en particulier les chapitres 11 et 14.

7 Nous parlerons rapidement dans le chapitre 7 consacré à la résolution en calcul des prédicats des résultats démontrés par Herbrand : domaines d'Herbrand et théorème d'Herbrand.

3.5 Propriétés fondamentales

Nous allons dans ce chapitre nous intéresser rapidement aux propriétés de consistance syntaxique et de complétude du calcul des prédicats. Rappelons qu'il s'agit d'étudier l'équivalence entre théorie des modèles et théorie de la démonstration *i.e.*, entre $\vdash B$ et $\models B$.

3.5.1 Adéquation - Consistance

Le théorème d'adéquation du calcul propositionnel se généralise assez simplement au calcul des prédicats (la démonstration fut réalisée pour la première fois par Hilbert et Ackermann). Elle ne pose pas de difficultés majeures.

Théorème 3.2 – *En calcul des prédicats, si* $\vdash B$ *alors* $\models B$. *Le calcul des prédicats est donc adéquat.*

Notons que le calcul des prédicats est aussi consistant, pour les mêmes raisons que le calcul propositionnel (il est adéquat pour une « bonne » théorie des modèles).

3.5.2 Complétude

Le théorème de complétude du calcul des prédicats fut démontré beaucoup plus tardivement, et parallèlement aux recherches sur la complétude de l'arithmétique.

Sa démonstration est totalement différente de celles que nous avons pu évoquer jusqu'ici. Nous nous contenterons de l'admettre (voir (Kleene 1987; Enderton 1972) pour une démonstration complète).

Théorème 3.3 – *En calcul des prédicats, si* $E \models B$ *alors* $E \vdash B$. *Le calcul des prédicats est donc fortement complet.*

Il n'est pas syntaxiquement complet. Tout comme pour le calcul propositionnel, cela découle de la complétude forte et de l'existence de formules contingentes telles que la formule $\exists x\, P(x) \rightarrow \forall x\, P(x)$ qui est vraie pour certaines interprétations de P, fausse pour d'autres.

3.5.3 Décidabilité

Le calcul des prédicats est indécidable. Ce résultat fondamental fut démontré tardivement. Nous en reparlerons dans le chapitre 5. Il est plus précisément *semi-décidable* puisque toute formule valide est un théorème *i.e.*, la recherche d'une preuve pour une formule valide s'effectuera certainement en un temps fini. Par contre, la recherche d'une preuve pour une formule non valide peut ne pas s'arrêter.

3.5.4 Remarques

Nous n'avons pas, dans ce chapitre, introduit de notions fondamentalement nouvelles. Les définitions de consistance, d'adéquation, de complétude restent les mêmes, et les méthodes utilisées pour étudier la logique considérée (théorie des modèles et théorie de la démonstration) sont également semblables.

Nous commençons cependant à connaître les premières difficultés concernant la décidabilité, difficultés qui seront, pour la logique mathématique, le début du doute.

CHAPITRE 4

Les machines de Turing

Jean-Marc Alliot

4.1 Introduction

Dans les années 1930-1936, la communauté mathématique travaillait à définir la notion de calculabilité et plus particulièrement de calculabilité intuitive. Grossièrement, les mathématiciens cherchaient à savoir ce que l'homme estimait être une fonction mathématique que l'on pouvait calculer, au sens pratique du terme.

Certains résultats avaient déjà été obtenus par Kleene et Church en 1932-1935 : ils avaient défini une classe de fonctions, les fonctions *λ-calculables*, qui, d'après eux, représentaient la classe des fonctions intuitivement calculables. Ces travaux allaient donner naissance au *λ-calcul* qui reste une base de l'informatique théorique (cf. chapitre 12).

En 1934, Kurt Gödel présenta une autre classe de fonctions, les fonctions *récursives générales*. Cette classe représentait également à ses yeux la classe des fonctions intuitivement calculables. Il fut démontré par Church et Kleene en 1936 que les fonctions λ-calculables et les fonctions récursives générales recouvraient la même classe de fonctions calculables *i.e.*, toute fonction récursive générale était λ-calculable et réciproquement.

L'inconvénient de ces deux présentations étaient leur manque de signification intuitive. C'est en 1936 que le mathématicien anglais Turing[1] publia un article décrivant la classe des fonctions appelées ultérieurement *Turing-calculables*. Comme nous allons le voir une fonction *Turing-calculable* est une fonction pour laquelle on est capable de construire une description algorithmique de calcul à destination de *machines de Turing*.

---

1 Alan Mathison Turing (1912-1954) fut un des grands mathématiciens de son époque. Il se consacra aux problèmes de métamathématique et d'informatique théorique, au même titre que Gödel, Church, Kleene, Herbrand, Skolem, Post, Von Neumann... Il participa pendant la seconde guerre mondiale à l'opération ENIGMA qui avait pour but de décoder les messages secrets allemands (la théorie de la cryptographie a d'ailleurs des rapports importants avec l'informatique théorique). C'est à cette occasion (1943) que fut réalisé le premier calculateur électronique, le COLOSSUS (quoique certains estiment en fait que le premier modèle de calculateur électronique a été développé en 1935-1940 par le physicien américain John Atanassof (Mackintosh 1988). La question de savoir quel fut le premier « ordinateur » est complexe. On peut lire pour plus d'informations le commentaire fait par René Moreau de l'article sus-nommé ou pour une vision plus large des choses (Breton 1987)). Turing fut probablement un des mathématiciens qui entrevit le premier ce que pourrait devenir l'informatique. Dans un de ses articles, il énonça tout un ensemble de principes pour construire des machines qui jouent aux échecs ; dans un autre il définit le *test de Turing* dont nous avons parlé dans le chapitre 1. Personnalité étrange, peu adapté à l'Angleterre encore puritaine dans laquelle il vivait, il se suicida à l'âge de 42 ans.

En ce sens, les machines de Turing peuvent être considérées comme les précurseurs des ordinateurs modernes, et les programmes de *machines de Turing* sont les précurseurs des langages procéduraux[2] de programmation classique[3].

En 1937, Turing démontra l'équivalence des fonctions Turing-calculables et récursives générales. Il énonça alors avec Church, la thèse suivante, dite *thèse de Church-Turing* :

Définition 4.1 – Thèse de Church-Turing – *Toute fonction intuitivement calculable est Turing-calculable.*

Il nous faut bien garder en mémoire que ce principe n'est qu'une hypothèse qu'il est impossible de démontrer, la notion de calculabilité intuitive étant bien difficile à déterminer.

Cette thèse serait évidemment infirmée s'il était possible de présenter une fonction calculable « intuitivement » qui ne soit pas Turing-calculable.

Le fait que 50 ans après, on n'ait jamais pu trouver de semblables fonctions incite à penser que cette hypothèse a une bonne chance d'être exacte. D'autre part, certains résultats théoriques (théorème de récursion de Kleene), montrent qu'une fonction contredisant la thèse de Church-Turing devrait être construite de telle façon qu'elle ne ferait appel à aucune fonction dont nous savons qu'elle est Turing-calculable, c'est-à-dire à aucune fonction que nous connaissons aujourd'hui. Il semble difficile de construire intuitivement une telle fonction.

Pourquoi étudions-nous les machines de Turing ? Tout d'abord parce qu'elles représentent une des bases fondamentales de l'informatique théorique, mais aussi pratique. Ensuite parce qu'elles vont nous permettre de donner des définitions simples de la calculabilité et sont indispensables pour étudier les problèmes de complexité. Enfin parce que la démonstration du théorème d'incomplétude de Gödel par les machines de Turing (donnée par Kleene en 1943) est particulièrement élégante et montre les rapports étroits entre métamathématique, logique, informatique théorique et intelligence artificielle.

4.2 Machine de Turing déterministe

Comme nous allons le voir dans ce paragraphe, la définition d'une machine de Turing[4] est celle d'un calculateur numérique idéal.

Définition 4.2 – Structure de stockage – *Une machine de Turing opère sur une bande linéaire divisée en une infinité de cases. Chacune de ses cases pourra, à un instant donné, contenir un symbole pris dans un ensemble fini de symboles* $\Gamma = \{s_0, s_1, \ldots, s_n\}$ *(appelé alphabet). L'ensemble des mots*[5] *sur* Γ *sera noté* Γ^*. *Le symbole* s_0 *est traditionnellement le* blanc, *que nous noterons aussi* B. *À l'instant initial, toutes les cases, sauf éventuellement un*

2 Même s'il n'est historiquement pas évident que les machines de Turing aient réellement influencé le développement des langages classiques.

3 Le λ-calcul fut à la base des langages fonctionnels (LISP, cf. chapitre 17). La logique mathématique engendra la programmation logique (PROLOG, cf. chapitre 18).

4 Pour plus de détails, on peut se reporter à (Minsky 1967; ven Emde Boas 1990).

5 Un mot est une suite finie de symboles de Γ. Nous reviendrons de façon plus précise sur les alphabets et les mots dans le chapitre consacré aux langages formels.

nombre fini, contiennent s_0. *Une machine de Turing à un instant donné n'a accès qu'à une seule de ces cases à travers une* fenêtre de lecture.

Définition 4.3 – États – *Une machine de Turing se trouve à un instant donné dans un état* z_j *pris dans un ensemble* $Z = \{z_0, z_1, \ldots, z_m, z_h\}$ *fini. On distinguera en particulier l'état* z_0, *dit état de démarrage (ou état initial) et l'état* z_h *dit état d'arrêt.*

Intuitivement les états correspondent à l'état de l'unité centrale d'un calculateur, alors que la bande et l'alphabet correspondent à la structure de la mémoire.

Définition 4.4 – Fonction de transition – *La fonction de transition d'une machine de Turing est une application* δ *définie par :*

$$\delta : ((Z - \{z_h\}) \times \Gamma) \rightarrow (Z \times \Gamma \times \{G, D, I\})$$

Ainsi $\delta(z, s) = (z', s', G)$ signifie que si la machine de Turing est dans l'état z et lit une case contenant le symbole s elle passera dans l'état z', écrira dans la case le symbole s' à la place de s et ira sur la case de gauche (D signifie que la machine va à droite et I qu'elle reste immobile).

Si la machine aboutit à l'état z_h, cela signifie que la machine a terminé son calcul. On introduit parfois, sans que cela affecte la puissance de la machine, plusieurs états d'arrêt pour représenter, indépendamment du contenu de la bande lors de l'arrêt, plusieurs résultats possibles[6].

Intuitivement, la fonction de transition correspond à la notion de programme ou d'algorithme dans un calculateur traditionnel. On appelle parfois le graphe de la fonction de transition le programme de la machine de Turing.

Définition 4.5 – Machine de Turing – *Une machine de Turing est définie par le n-uplet* $(\Gamma, Z, \delta, z_0, z_h)$.

Définition 4.6 – Configuration d'une machine de Turing – *La configuration d'une machine de Turing est un élément de* $\Gamma^* \times (Z \times \Gamma) \times \Gamma^*$, *donc un triplet* $(u, (z, s), w)$. u *représente le mot à gauche de la tête de la machine de Turing,* w *le mot à droite de la tête de la machine de Turing,* z *est l'état de la machine de Turing et* s *le symbole inscrit dans la case lue par la tête de lecture.*

Une configuration d'arrêt *est un triplet* $(u, (z, s), w)$ *tel que* $z = z_h$.

Le comportement d'une machine de Turing est entièrement déterminé dès que l'on connaît $(\Gamma, Z, \delta, z_0, z_h)$ et la configuration C courante.

Définition 4.7 – Configuration initiale définie par un argument – *Soit le mot* $m = (a_1 a_2 \ldots a_n) \in \Gamma^*$. *On appelle configuration initiale définie par* m *la configuration* $(B, (z_1, a_1), a_2 a_3 \ldots a_n B)$. *($B$ signifie que l'on ne trouve plus que des blancs.)*

Définition 4.8 – Séquence de calcul – *On dit qu'une configuration* C' *succède à une configuration* C *et l'on note* $C \rightsquigarrow C'$ *si la machine de Turing dans la configuration* C *passe*

---

6 En particulier, l'introduction de deux états d'arrêt z_y et z_n est fréquente lors du calcul d'une fonction à valeur dans $\{t, f\}$ par exemple. Ce sera en particulier le cas lors de l'étude de la complexité des problèmes de décision dans le chapitre 11.

dans la configuration C' en appliquant sa fonction de transition. Une séquence de calcul est une succession $(C_0, C_1, \ldots, C_k)$ de configurations telle que $\forall i, 0 \leq i \leq k-1$, $C_i \rightsquigarrow C_{i+1}$.

Définition 4.9 – Mot accepté par une machine de Turing – *Soit un mot m et la configuration initiale définie par m : $C(m)$. On dit que le mot m est accepté par la machine de Turing $\mathcal{M}$ s'il existe un $k \in \mathbb{N}$ tel que*

$$C(m) \rightsquigarrow C_1 \rightsquigarrow \ldots \rightsquigarrow C_k$$

et C_k est une configuration d'arrêt. Un tel calcul est appelé calcul d'acceptation *de m.*

Définition 4.10 – Langage accepté par une machine de Turing – *L'ensemble des mots m de Γ^* tel que m est accepté par la machine de Turing $\mathcal{M}$ est appelé langage accepté par $\mathcal{M}$. On notera ce langage $L(\mathcal{M})$.*

Une machine de Turing est donc munie d'une capacité mémoire infinie. Il faut d'autre part remarquer que l'ensemble des symboles $\{s_0, s_1, \ldots, s_n\}$ et l'ensemble des états $\{z_0, z_1, \ldots, z_m, z_h\}$ étant l'un et l'autre finis, la définition d'un programme de machine de Turing est nécessairement finie.

Les machines de Turing que nous venons de définir sont appelées *machines de Turing déterministes*. En effet, dans une configuration donnée, la machine de Turing $\mathcal{M}$ peut effectuer au plus une transition (donnée par δ). En partant d'une configuration donnée, elle effectuera donc toujours la même séquence de calcul (séquence qui peut d'ailleurs ne jamais se terminer).

Il existe bien d'autres définitions des machines de Turing déterministes (ou d'autres modèles) que l'on peut trouver chez quantité d'auteurs (Kleene 1987; Salomaa 1989; Johnson 1990; van Emde Boas 1990; Garey and Johnson 1979). Il peut s'agir de machines à plusieurs bandes, ou une bande de longueur finie d'un côté, avec un alphabet de deux symboles, ou une définition des états de démarrage et d'arrêt différente de celle que nous donnons. Il faut savoir que toutes ces machines de Turing sont équivalentes au sens de la calculabilité.

4.3 Notion de calcul

4.3.1 Calcul d'un argument et fonction calculable

Définition 4.11 – Calcul d'un argument – *On dit qu'une machine de Turing calcule* la valeur $b \in \Gamma^*$ *pour l'*argument $a \in \Gamma^*$ *si :*

- *a est accepté par $\mathcal{M}$*
- *lorsque la machine de Turing atteint son état d'arrêt, le mot représenté par la bande est b.*

Le temps d'acceptation étant fini, et une écriture ayant lieu par transition, b est nécessairement fini. La notion de calcul d'un argument par une machine de Turing est bien adéquate à l'idée intuitive que avons de la notion de calcul. Remarquons que la notion de finitude du temps de calcul (du nombre fini d'étapes dans le calcul d'acceptation) est

	B	0	1
z_0	(z_1, B, G)	$(z_0, 0, D)$	$(z_0, 1, D)$
z_1	$(z_h, 1, I)$	$(z_h, 1, I)$	$(z_1, 0, G)$

Table 4.1 – Programme de la machine de Turing effectuant $f(x) = x + 1$

fondamentale dans la définition. Si nous construisons un programme de machine de Turing dont nous démontrons qu'il calculera un résultat pour un argument a sans pouvoir démontrer que ce résultat sera obtenu en un temps fini nous ne dirons pas que la machine de Turing est capable de calculer un résultat lorsqu'elle est appliquée à a.

Définition 4.12 – Fonction calculable – *On dit qu'une fonction f définie sur un domaine $\mathcal{D}$ est Turing-calculable*[7] *si nous pouvons construire une machine M telle que M appliquée à tout élément a de $\mathcal{D}$ calcule $f(a)$.*

Notons tout de suite que la définition impose à la machine de calculer la valeur $f(a)$ pour toute valeur de a de l'ensemble $\mathcal{D}$ sur lequel f est définie, et ce, même si $\mathcal{D}$ est infini.

Remarquons également qu'en raison de la thèse de Church-Turing, les notions de Turing-calculabilité et de calculabilité intuitive sont considérées comme étant équivalentes. On parlera donc de *fonction calculable* au lieu de *fonction Turing-calculable.*

4.3.2 Exemple : $f(x) = x + 1$

Nous allons rapidement montrer que $f(x) = x+1$, définie sur l'ensemble des entiers strictement positifs, est calculable. Pour ce faire, nous allons construire une machine de Turing calculant la fonction f.

Le premier point auquel nous devons nous arrêter est l'alphabet de notre machine. Il s'agit ici du système de représentation numérique que nous allons décider d'adopter. De nombreux auteurs choisissent l'alphabet minimal $\{B, |\}$. Dans cet alphabet, un nombre x est représenté par x bâtons sur la bande. Cette représentation est appelée *représentation unaire.* Ce type de représentation n'est pas admissible en théorie de la complexité (nous y reviendrons dans le chapitre 11), en revanche elle est très utilisée par les logiciens. D'autres adoptent le système de numération décimale, ce qui les amène à considérer l'alphabet $\{B, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

En ce qui nous concerne, nous adopterons, pour cet exemple, le système de numération binaire ($\Gamma = \{0, 1, B\}$) qui représente un intéressant moyen terme entre les systèmes précédents (une machine de Turing opérant sur un alphabet minimal a souvent un grand nombre d'états, alors que le système de numération décimale comporte un nombre de symboles important). Un entier sera codé par sa représentation en base 2.

Le programme de notre machine est représenté table 4.1.

Considérons par exemple la ligne correspondant à l'état z_1. Elle signifie que si la machine est dans l'état z_1 et que le symbole courant est :

---

7 Cette définition présuppose l'existence d'une fonction de codage qui permet de transformer tout élément de $\mathcal{D}$ en un mot de Γ^*, et d'une fonction de décodage qui effectue le travail inverse.

Un blanc La machine écrit un 1, ne bouge pas et passe dans l'état z_h.

Un 0 La machine écrit un 1, ne bouge pas et passe dans l'état z_h.

Un 1 La machine écrit un 0, se déplace à gauche et reste dans l'état z_1.

Quel est le principe de fonctionnement de cette machine ? Partant de la gauche (premier symbole non blanc) de son argument, elle va vers la droite en restant dans l'état z_0 jusqu'au moment où elle rencontre un symbole blanc. Elle passe alors dans l'état z_1 et revient à gauche. La machine ne sortira de cet état z_1 que lorsqu'elle rencontrera une case 0 ou B lui permettant de terminer son addition, en inscrivant un 1 et en passant dans l'état d'arrêt z_h. Tant qu'elle lira des 1, elle les remplacera par des 0 et passera à gauche.

On constate donc que cette machine calcule la représentation binaire de $x+1$ à partir de la représentation binaire de n'importe quel $x \in \mathbb{N}$. Donc $f(x) = x + 1$ est calculable.

4.4 Réalisation pratique

L'exemple précédent montre qu'il n'est déjà pas immédiat de construire une machine de Turing calculant la fonction $x + 1$.

On peut alors se demander si notre notion de calculabilité intuitive est bien valable, car comment prouver qu'une fonction est calculable si l'on est en pratique incapable de construire la machine de Turing associée ?

Il existe deux réponses à cet argument.

La première est simplement de dire que nous ne demandons pas, pour dire qu'une fonction est calculable, de construire explicitement une machine de Turing, mais simplement de démontrer qu'il existe une machine de Turing calculant la fonction.

Pour ceux qui, membres de l'école constructiviste, refusent ce genre de réponse, signalons qu'il existe des méthodes permettant de construire automatiquement des machines de Turing à partir des fonctions récursives générales, et qu'il est actuellement possible de donner des méthodes permettant de construire de façon automatique une machine de Turing pour toutes les fonctions intuitivement calculables connues.

En fait, dans toute la suite de cet ouvrage, nous ne chercherons jamais à calculer explicitement une machine de Turing et nous nous contenterons de montrer par des arguments intuitifs qu'une telle fonction existe.

4.5 Les machines de Turing non déterministes

Comme nous l'avons vu, une machine de Turing dans un état donné lisant un symbole donné, peut effectuer au plus une action : celle qui figure dans son programme. En revanche une machine de Turing non déterministe peut, dans un état donné et face à un symbole donné, choisir une action parmi plusieurs. Chaque case dans le tableau d'une machine de Turing est remplacée par une liste d'actions possibles. Pour explorer complètement l'ensemble des exécutions possibles, il faut donc effectuer, l'un après l'autre, l'ensemble des choix possibles. Au lieu d'être séquentiel, le schéma d'exécution devient arborescent. Chaque nœud de l'arbre correspond à une configuration et est un point de

choix pour la machine de Turing, chaque segment correspond à une transition, chaque feuille à une position terminale de la résolution (une configuration d'arrêt) et chaque branche à une suite de choix.

Formellement, la différence se situe au niveau de la fonction δ dont l'espace d'arrivée n'est plus $\Delta = (Z \times \Gamma \times \{G, D, I\})$, mais l'ensemble des n-uplets (*n fini*) d'éléments de Δ.

Les machines de Turing non déterministes sont peu utilisées par les logiciens. En revanche, elles ont une grande importance en théorie de la complexité, c'est la raison pour laquelle nous les avons brièvement définies ici.

4.6 Remarques

Notons tout de suite que, de par la définition de la calculabilité, une fonction $f(a)$ n'est calculable que s'il existe une machine de Turing pouvant calculer f pour *toute* valeur de a. En particulier, une machine de Turing doit être capable d'effectuer des calculs sur des nombres arbitrairement grands. C'est parce que les fonctions calculables doivent être totalement calculables, que nous pourrons bientôt construire une fonction non-calculable.

Remarquons également la similitude qui existe entre une machine de Turing et un humain effectuant un calcul *numérique*. Comme le fait remarquer Kleene (Kleene 1987), on peut considérer la bande de la machine de Turing comme une feuille de papier, les symboles utilisés sont les symboles de calcul et les états de la machine sont les « états mentaux » du calculateur humain, états qui lui permettent de se rappeler à quel point du calcul il se trouve à un instant donné. Le nombre d'états du calculateur (humain ou mécanique) est supposé fini. Citons la phrase de Turing :

> « Le nombre d'états mentaux qu'il faut prendre en considération est fini. Si nous admettions une infinité de ces états, certains d'entre eux seraient aussi voisins que l'on voudrait et il y aurait des confusions. »

Il s'agit là, bien entendu, d'un postulat qui prétend qu'un être humain effectue des séries de calculs de façon numérique et non analogique. Si cette hypothèse est peu critiquable quand elle s'applique à des nombres, elle est loin d'être évidente si l'on se réfère à des domaines plus complexes comme la compréhension du langage ou la perception des formes. Les textes de Turing ayant été une des bases du développement de l'IA, on comprend mieux les thèses et postulats mécanistes développés quelques années plus tard en s'appuyant précisément sur ces paroles.

CHAPITRE 5

Les systèmes formels

Jean-Marc Alliot

5.1 Un peu d'histoire

À compter de la Renaissance, en passant par le Siècle des lumières et la révolution française, et jusqu'au début du vingtième siècle (approximativement), la Science est devenue la principale croyance, voire religion déguisée, de l'humanité. Nous profitons encore, anciens élèves d'écoles d'ingénieurs, d'universités... des retombées de ce formidable mouvement. Cependant, depuis une cinquantaine d'années, la Science tend à perdre la place dominante qu'elle occupait, au profit de l'économie et du « management ».

Il n'est guère besoin de longs discours pour établir le phénomène, les faits parlent d'eux-mêmes. Dans la littérature populaire, le personnage central souvent campé par le savant au dix-neuvième siècle[1], est de plus en plus occupé par un (ou même souvent *une*, là aussi signe des temps) jeune « raider », aventurier de Wall-Street et du billet vert.

Au niveau philosophique, la Mathématique fut longtemps considérée comme une référence absolue de justesse, une quasi-manifestation de l'esprit divin[2].

Il y a une centaine, voire une cinquantaine d'années, les physiciens étaient des personnages publiquement connus (Maxwell, Edison, Einstein, Bohr, et bien d'autres furent célèbres de leur temps). Aujourd'hui, les gens célèbres sont les grands financiers qu'il soient triomphants ou déchus mais quel homme de la rue serait capable de citer le nom d'un grand savant contemporain ?

Enfin, nombre de postes de direction de grandes entreprises ou de grands services, confiés traditionnellement à des gens de culture scientifique, vont de plus en plus vers des spécialistes de « management », de gestion et d'économie[3].

---

1 Voir Jules Verne ou surtout Gustave Le Rouge, feuilletonniste de journaux populaires.

2 Il est intéressant de relire (Kant 1944) pour voir la place que Kant accorde à la science mathématique : « Les inévitables problèmes de la *Raison Pure* sont *Dieu*, la *liberté*, l'*immortalité* et la science, qui avec tous ses procédés, n'a pour but final que la solution de ces problèmes (...) Une partie de ces connaissances, la Mathématique, possède de longue date la certitude et donne par là bon espoir aussi pour les autres. »

3 Ce n'est pas une règle absolue, bien sûr, mais un phénomène qui se reproduit régulièrement.

Les raisons de ce changement sont multiples ; certaines sont structurelles et tiennent à l'évolution profonde de la société plus orientée aujourd'hui vers le paraître et le posséder que vers l'être et le faire[4].

Mais il y a des raisons liées à l'évolution de la science elle-même ; tout d'abord la science est devenue extrêmement complexe, spécialisée et difficile à vulgariser. Il est plus facile d'impressionner quelqu'un en réalisant la première transmission par voie Hertzienne qu'en montrant le résultat d'une expérience en chambre à bulles prouvant l'existence d'une nouvelle particule élémentaire. Les chercheurs ne travaillent plus individuellement mais en équipe et les théories scientifiques modernes sont très spécialisées, incompréhensibles pour le grand public : il était possible de donner une image intuitive de la relativité, mais essayez donc d'expliquer la théorie des super-cordes à un auditoire novice[5].

Enfin, la dernière raison qui nous intéresse au premier chef : la crise interne qu'ont traversée et traversent encore les deux sciences majeures des siècles derniers : la Mathématique et la Physique.

Nous ne dirons qu'un mot des problèmes de la physique contemporaine. Les gens intéressés pourront se référer pour plus de précisions à (d'Espagnat 1989) ou (Pomian 1990). Grossièrement, on peut dire que les difficultés de la physique sont liées à la remise en cause d'un principe sacré de la Science : celui du déterminisme. Un certain nombre d'expériences tendrait à prouver qu'il faut renoncer à ce principe ou au dogme voulant qu'il n'y ait pas d'interférences pouvant se déplacer plus vite que la lumière. Le débat fait encore rage en ce moment. Mais avant même que ces problèmes se posent, les premiers résultats « paradoxaux » de la mécanique quantique (impossibilité de connaître précisément à la fois la position et la vitesse d'une particule, caractère dualiste (ondulatoire et corpusculaire) de la notion de particule, ...) avaient fortement secoué la communauté scientifique[6], séisme qui s'était propagé jusque dans la communauté sociale en général.

Les problèmes qu'affrontent les mathématiques ne sont pas moins intéressants et ce chapitre va permettre, je l'espère, de mieux les comprendre.

---

4 Se repose à ce propos le problème classique : l'évolution de l'attitude de la société vis-à-vis de la science vient-elle de l'évolution de la société ou au contraire l'évolution de la société est-elle une conséquence de l'évolution de son appréhension de la science ?

5 « Depuis, disons un demi-siècle, la masse des connaissances acquises, dans presque tous les domaines, a grandi comme on sait à peu près au delà de toute expression — il n'est plus question, et depuis longtemps, pour un cerveau normalement conditionné, d'en tenir registre, et de s'en faire une idée lointaine autrement qu'à travers des vulgarisations, non plus de seconde, mais de troisième ou quatrième main » (Gracq 1950).

6 On peut rappeler la célèbre remarque d'Einstein relative à certains résultats de la mécanique quantique : « Je refuse de croire que Dieu joue aux dés avec le cosmos ». Einstein fut d'ailleurs à l'origine de la théorie des *variables cachées* qui prétendait que certains paramètres que nous ne pouvons connaître en mécanique quantique (comme connaître simultanément la position et la vitesse d'une particule) nous étaient inaccessibles par manque d'information, et non comme le prétendait Bohr, parce qu'il était impossible théoriquement de les connaître. Le physicien anglais John Bell découvrit une formulation (les inégalités de Bell) permettant la vérification expérimentale de la théorie d'Einstein. Les dernières expériences semblent prouver qu'Einstein avait tort et Bohr raison. Pour un développement de l'attitude philosophique d'Einstein relativement à la physique, on peut lire les excellents essais de Gerard Holton (Holton 1982a; 1982b).

5.2 Les problèmes des mathématiques

Nous allons nous attacher à montrer dans la suite de ce paragraphe l'ampleur de la crise des mathématiques au XXe siècle. Il n'existe sans doute pas de meilleures phrases pour décrire cette crise que celles de Morris Kline dans son introduction (Kline 1989) :

> « Il y a des tragédies provoquées par la guerre, la famine et la peste. Mais il existe aussi des tragédies intellectuelles provoquées par les limites de l'esprit humain. Ce livre est en quelque sorte le récit des calamités qui ont mis bas la plus efficace et incomparable réalisation de l'homme, en son effort permanent et approfondi de la raison humaine : les mathématiques. »

5.2.1 Les malheurs de la géométrie euclidienne

Jusqu'au dix-neuvième siècle, les mathématiciens connurent fort peu de problèmes métaphysiques. Certes, on se posait des questions sur l'équivalence des différentes branches des mathématiques et René Descartes démontra très tôt l'équivalence entre la géométrie traditionnelle et la théorie analytique. Mais on ne doutait en aucun cas de la validité des axiomes de la géométrie euclidienne[7], pierre d'angle des mathématiques de l'époque. Le premier sérieux coup à cette confiance fut donné par Bolyai, Lobachevsky et Riemman qui démontrèrent l'indépendance du troisième axiome d'Euclide (par un point on ne peut faire passer qu'une parallèle à une droite), et le remplacèrent soit par un axiome prétendant qu'on pouvait faire passer par ce point une infinité de droites, soit par un axiome prétendant qu'on ne pouvait en faire passer aucune[8].

À partir de ce moment, commencèrent à se faire jour les notions d'axiomatique matérielle et d'axiomatique formelle, et les risques que font courir la confusion de ces deux notions ; la géométrie euclidienne classique était une théorie axiomatique matérielle : chaque axiome avait une signification intuitive immédiate, et les termes employés avaient une réalité physique « obligatoire » dans l'esprit de chacun : une droite géométrique était une droite « physique ». Bolyai, Lobachevsky et Riemman en firent (probablement sans le savoir) un système d'axiomatique formelle : ils reprirent les axiomes, supprimèrent les notions intuitives placées implicitement derrière les mots et purent ainsi modifier un axiome. La notion de droite et de point dans leurs systèmes n'avaient plus la signification intuitive primitive et représentaient des objets formels sur lesquels on opérait mentalement. Certes, il existe des modèles (la géométrie sphérique pour Riemman, ou la géométrie des surfaces à courbure négative pour Bolyai-Lobachevsky), où ces systèmes formels reprennent une interprétation intuitive. Mais ce ne sont alors que des modèles d'application de la théorie formelle.

Ces résultats sonnaient les premiers le glas de la géométrie traditionnelle, qui perdait tout fondement « métaphysique », et aussi des mathématiques traditionnelles. Les systèmes formels étaient en route et avec eux, ce que l'on a coutume d'appeler aujourd'hui les mathématiques modernes.

---

7 Voir plus haut la remarque concernant E. Kant.

8 Il s'agit là de la première application d'une technique devenue classique qui consiste à montrer l'indépendance d'un axiome $\mathcal{A}$ en construisant une théorie consistante comprenant tous les axiomes, sauf $\mathcal{A}$ que l'on remplace par un axiome différent et contradictoire avec le précédent.

5.2.2 L'infini

L'infini commença à poser des problèmes aux mathématiciens très tôt. Leibnitz, lors de ses travaux sur le calcul différentiel, rencontra cet objet bizarre. Les sentiments de la communauté mathématique furent plutôt circonspects voire défavorables. On cite souvent la phrase de Gauss (1831) : « Je m'élève contre l'emploi de grandeurs infinies conçues comme des réalités achevées car cela n'est jamais permis en mathématique ». Cette phrase visait la notion de grandeur infinie (le $+\infty$ que nous employons aujourd'hui), mais un autre problème de l'infini se posait aussi, celui des ensembles infinis.

En 1638, Galilée remarquait le « paradoxe » suivant : il est possible de mettre en correspondance chaque élément de l'ensemble des entiers naturels avec un et un seul élément de l'ensemble des carrés d'entiers naturels[9]. Or la méthode de comptage par correspondance un à un est sans doute la méthode la plus intuitive qui soit. Deux enfants qui ne savent pas compter, peuvent déterminer quel est celui d'entre eux qui a le plus de billes : il suffit qu'ils en posent chacun simultanément une jusqu'à ce qu'il n'y ait plus qu'un seul enfant à avoir des billes dans la main. Le résultat de Galilée prouvait-il donc qu'il y a « autant » de carrés qu'il y a d'entiers ? Et que signifiait dans ce contexte le mot *autant*, car intuitivement il semble bien qu'il y ait plus d'entiers que de carrés parfaits.

5.2.3 Les ensembles

Il faudra attendre le dix-neuvième siècle et le mathématicien Georg Cantor pour « formaliser » les notions d'ensemble et d'infini. Cantor imposa comme définition de *autant* : « peut être mis en correspondance un à un avec » (dans la terminologie moderne, nous dirions qu'il existe une bijection entre les deux ensembles, ou que les deux ensembles sont équipotents (Xuong 1992)). Ainsi, l'ensemble des entiers et l'ensemble des carrés d'entiers avaient bien « autant » d'éléments. Les ensembles qui peuvent être mis en correspondance un à un avec l'ensemble des entiers naturels[10] sont dits *dénombrables.*

Cantor démontra rapidement que l'ensemble des fractions rationnelles était lui aussi dénombrable, ainsi que l'ensemble des nombres algébriques[11]. Se posa alors le problème de savoir s'il existait des ensembles non dénombrables. Cantor s'intéressa à l'ensemble des fonctions définies partout sur l'ensemble des entiers naturels[12] *i.e.*, les fonctions de $\mathbb{N}$ dans $\mathbb{N}$. Cantor démontra que cet ensemble n'était pas dénombrable, en utilisant une méthode, dite *méthode de la diagonale cantorienne*[13], que nous allons étudier plus en détail ; nous aurons en effet à la réutiliser dans les paragraphes suivants pour démontrer la non-calculabilité de certaines fonctions.

La démonstration de Cantor est une démonstration par l'absurde. Il suppose qu'il existe une correspondance un à un entre l'ensemble de fonctions sur $\mathbb{N}$ et l'ensemble des entiers naturels. On peut alors numéroter ces fonctions à l'aide de cette correspondance : $f_0, f_1, f_2, \ldots, f_n \ldots$ On définit alors la fonction :

---

9 $1 \leftrightarrow 1, 2 \leftrightarrow 4, 3 \leftrightarrow 9, \ldots$

10 Cet ensemble sera souvent noté $\mathbb{N}$ dans la suite de l'énoncé. Nous utiliserons aussi $\mathbb{N}$ pour désigner le système formel de l'arithmétique. Il y a des raisons historiques à la confusion des notations.

11 Nombres solutions d'une équation polynomiale à coefficients entiers.

12 La partie entière de la racine carrée, la valeur absolue des fonctions polynomiales,...

13 Cette méthode est connue de tous les anciens élèves de classe préparatoire, mais il est parfois bon de rafraîchir les mémoires ...

	0	1	2	...
f_0	$f_0(0)$	$f_0(1)$	$f_0(2)$	...
f_1	$f_1(0)$	$f_1(1)$	$f_1(2)$	...
f_2	$f_2(0)$	$f_2(1)$	$f_2(2)$	...
...	...	...	...	...

Table 5.1 – Diagonale cantorienne

$$f(i) = f_i(i) + 1$$

La fonction f que nous venons de construire est une fonction partout définie sur $\mathbb{N}$. Donc elle doit appartenir à notre énumération de fonctions. Donc, il doit exister un nombre n tel que pour tout i :

$$f_n(i) = f(i)$$

Mais considérons alors l'égalité précédente en prenant pour valeur de i précisément n :

$$f_n(n) = f(n) =_{def} f_n(n) + 1$$

Ceci est évidemment impossible, donc il n'existe pas de correspondance un à un entre l'ensemble des fonctions partout définies sur $\mathbb{N}$ et l'ensemble $\mathbb{N}$. Pourquoi appelle-t-on cette méthode *diagonale cantorienne* ? On peut le voir sur le tableau 5.1. La méthode de Cantor consiste à prendre chaque élément de la diagonale en lui ajoutant 1 pour définir sa fonction f. Par une méthode analogue, il est possible de démontrer que l'ensemble des fonctions de $\mathbb{N}$ dans $\{t, f\}$ ou que l'ensemble des nombres réels ne sont pas dénombrables[14].

Fort de ces résultats, Cantor se lança dans une théorie générale des ensembles abstraits. La définition de Cantor laisse cependant une large part à l'intuition : « Par un ensemble, j'entends toute collection M d'objets bien distincts m de notre perception ou de notre pensée ». Nous allons voir que le flou laissé autour de la définition d'ensemble va amener nombre de paradoxes.

5.2.4 Les paradoxes

Un des paradoxes les plus célèbres est sans doute celui de Russell (Russell 1989a). Russell considère l'ensemble E de tous les ensembles qui ne se contiennent pas eux-mêmes. La question que pose alors Russell est la suivante : E se contient-il lui-même ? Examinons successivement les deux possibilités :

---

14 Il est en fait facile de démontrer que $\text{Card}(\mathbb{R}) = \text{Card}(\mathbb{N} \to \mathbb{N})$. Cantor se posa alors la question suivante : est-il possible de trouver un ensemble E tel que $\text{Card}(\mathbb{R}) > \text{Card}(E) > \text{Card}(\mathbb{N})$? Ce problème, appelé *hypothèse du continu* , fut en partie résolu par Gödel qui démontra que si la théorie des ensembles était non contradictoire, on pouvait lui adjoindre l'hypothèse du continu et elle resterait non contradictoire. Paul Cohen résolut définitivement le problème en montrant qu'on pouvait aussi bien admettre l'hypothèse du continu (Gödel) que son contraire (1966).

- Si E se contient lui même, alors par définition il ne se contient pas lui-même, ce qui est contradictoire.
- Si E ne se contient pas lui-même, alors par définition il doit appartenir à E, ce qui est une fois de plus contradictoire.

Ce type de paradoxes découle du flou laissé par Cantor autour de sa définition d'ensemble. Il fut nécessaire de reformaliser sa théorie, en fait de construire une axiomatique formelle des ensembles abstraits. Zermelo, dès 1908, fut le premier à proposer de construire un tel système pour réduire la notion intuitive d'ensemble à une notion formelle. Une excellente axiomatique formelle des ensembles fut fournie par Gödel en 1940. Les lecteurs intéressés peuvent se reporter à (Kleene 1987). Cette classe de paradoxes est souvent désignée sous le nom de paradoxes logiques, dans la mesure où ils sont liés à la logique de la théorie en cause (ici la théorie ensembliste).

Les paradoxes sémantiques forment une toute autre classe de paradoxes. Un des plus célèbres est le paradoxe de Berry. On considère le nombre naturel défini par « le plus petit nombre entier ne pouvant être exprimé en moins de quinze mots ». On sait qu'il existe un ensemble des nombres naturels qu'on ne peut définir en moins de quinze mots. En raison des propriétés des sous-ensembles de $\mathbb{N}$, cet ensemble admet une borne inférieure I. Il semble même que ce nombre I soit 1297297 (un million deux cent quatre-vingt-dix-sept mille deux cent quatre-vingt-dix-sept, soit exactement quinze mots). Mais ce nombre I est aussi défini par l'expression « le plus petit nombre entier ne pouvant être exprimé en moins de quinze mots ». Or cette expression comporte *quatorze* mots. Il y a donc contradiction. Ce type de paradoxe est d'origine sémantique, c'est-à-dire lié au langage dans lequel nous exprimons la définition. En fait nous construisons là une définition autoréférente (cf. (Alliot and Schiex 1993), page 78) : nous utilisons un langage qui opère sur lui-même. Les paradoxes sémantiques attirèrent l'attention sur la nécessité d'utiliser des langages précis et restreints à un ordre de raisonnement. Nous retrouvons le problème que nous évoquions lors de l'introduction au chapitre 2 : celui du langage objet et du langage de l'observateur. Nous n'avons pas le droit d'utiliser le langage objet (d'ordre n) pour parler des objets, nous devons utiliser la langue de l'observateur (d'ordre $n+1$). De même si nous voulions parler du langage qui parle du langage qui parle de la logique, il nous faudrait définir un langage formel d'ordre $n+2$ (ou plutôt $(n+1)+1$). On peut citer comme autre genre de paradoxe sémantique le célèbre paradoxe du menteur : « la phrase que je dis est fausse », connu depuis l'Antiquité (Epiménide VIe siècle A.C).

La théorie des types de Russell a précisément été créée dans le but de formaliser correctement ce type de problèmes. Le but initial de Russell est la réduction des mathématiques à la théorie logique. Les *Principia Mathematica* publiés avec Whitehead de 1910 à 1913 constituent le travail qui fondera la logique moderne. Comme le signale D. Dubarle (Dubarle 1967), c'est Wittgenstein qui le premier remarquera le problème principal du réductionnisme de Russell (Wittgenstein 1961b) :

> « 6.1 Les propositions de la logique sont les tautologies.
>
> 6.31 La soi-disant loi d'induction ne peut en aucun cas être une loi logique, car elle est de toute évidence une proposition ayant un sens. »

L'axiome de l'infini de Russell (nécessaire à la démonstration de la loi d'induction) n'a pas le caractère d'évidence indispensable pour faire des *Principia Mathematica* un fondement valable de la mathématique, sur un plan philosophique.

Comme le fait d'ailleurs remarquer Kleene (1967) :

> « Le problème [des paradoxes] est plus profond [qu'on ne pourrait le croire] et nous contraint à nous demander sur quels points nous avons été induits en erreur par des méthodes de construction et de raisonnement qui avaient toujours paru convaincantes avant qu'on se fût avisé qu'elles aboutissaient à des paradoxes. L'unanimité entre les mathématiciens touchant les causes des paradoxes et les remèdes à y apporter n'a pas encore été atteinte et il est douteux qu'elle le soit jamais. »

L'ensemble des problèmes rencontrés par les mathématiques que nous venons de décrire dans ce chapitre semblait montrer qu'il était temps de renoncer aux mathématiques « intuitives » pour en venir à des constructions formelles pures, bien que la tentative logiciste formelle de Russell ait déjà montré quelques faiblesses. C'est dans ces conditions qu'éclata la controverse entre formalistes et intuitionnistes.

5.3 Intuitionnistes contre formalistes

5.3.1 Les intuitionnistes

Au lieu de tenter de résoudre les problèmes des mathématiques par des constructions formelles où l'intuition n'aurait plus sa place, les intuitionnistes souhaitaient revenir en arrière et ne conserver des mathématiques que la partie intuitivement accessible. Ils refusaient en particulier la notion d'infini achevé : pour eux, l'existence d'un ensemble achevé contenant tous les nombres naturels n'était pas évidente. Seul existait le fait, qu'on pouvait toujours construire un nombre plus grand que n'importe quel autre nombre. Tous les objets mathématiques devaient se construire[15].

Ce type d'attitude conduisit Brouwer[16] à contester certains principes considérés jusque-là comme évidents, comme le principe du tiers exclus[17]. Brouwer reconnaissait volontiers que dans le cas où le domaine $\mathcal{D}$ que parcourt un prédicat $P(x)$ est fini, $\exists x\, P(x) \vee \neg\exists\, xP(x)$ est vrai. Il est en effet possible dans ce cas de vérifier pour chaque élément de $\mathcal{D}$ si l'on a $P(x)$.

En revanche, pour un ensemble $\mathcal{D}$ infini, $\exists x\, P(x) \vee \neg\exists x\, P(x)$ n'est pas forcément vrai. Il nous est en effet matériellement impossible de vérifier pour tous les éléments de $\mathcal{D}$ si l'on a $P(x)$. Bien sûr, nous pouvons, avec de la chance, trouver un tel x rapidement, mais nous pouvons aussi bien ne jamais le trouver.

Il y a une autre conséquence des notions intuitionnistes : pour démontrer $\exists x\, P(x)$, il faut trouver un tel x et l'exhiber. Une démonstration par l'absurde, qui montrerait la fausseté de $\neg\exists x\, P(x)$ ne montrerait en aucun cas qu'un tel x existe mais seulement que $\neg\neg\exists x\, P(x)$ est vraie.

On voit donc que pour un intuitionniste, dans un ensemble infini, $\neg\neg A$ n'est pas systématiquement équivalent à A.

Depuis 1918, les mathématiques intuitionnistes se sont développées parallèlement aux mathématiques « traditionnelles ». Les démonstrations constructivistes donnent sou-

---

15 On confond parfois intuitionnisme et constructivisme, qui sont deux notions fort proches.

16 Qui fut le champion de l'école intuitionniste.

17 Rappelons que le principe du tiers exclus dit que pour tout P, $P \vee \neg P$ est vraie.

vent des résultats plus tangibles que les démonstrations formelles, mais sont plus difficiles. Il faut reconnaître que les intuitionnistes sont restés en deçà des résultats formalistes.

Leur apport fut néanmoins très important, surtout sur le plan épistémologique, car il conduisit à une réflexion profonde sur la façon d'appréhender l'infini.

5.3.2 Les formalistes

L'école formaliste emmenée par le mathématicien David Hilbert choisit une voie totalement différente de la voie intuitionniste.

Constatant que les paradoxes venaient de systèmes d'axiomes insuffisamment formalisés, Hilbert proposa de rédiger une théorie purement formelle des mathématiques et de démontrer sa consistance *formelle*. En effet, on avait jusque là toujours démontré les propriétés de consistance par rapport à des modèles : la géométrie de Bolyai-Lobachevsky avait été ramenée à la géométrie euclidienne, la géométrie euclidienne à l'analyse (Descartes)... Or ce type de démonstration montre seulement qu'une théorie est consistante si une autre théorie l'est. La propriété de consistance formelle demande simplement qu'à l'intérieur de la théorie formelle, on ne puisse avoir une formule qui soit démontrable et dont la négation soit démontrable (ce qui constituerait un paradoxe). Hilbert imposa également que les méthodes de démonstration de ce type de propriété soient « intuitivement convaincantes » et *finitistes*, c'est-à-dire ne faisant jamais appel à l'infini achevé. Hilbert appela cette nouvelle branche des mathématiques *métamathématique*. Hilbert énonça ce programme en 1908, et la réalisation commença vers 1920.

Hilbert se justifia de son attitude face à celle des intuitionnistes de la façon suivante : il scinda les mathématiques en deux domaines, celui des énoncés réels qui ont un sens intuitif et celui des énoncés idéaux qui n'en ont pas. Les énoncés idéaux ne sont que des moyens théoriques et élégants de simplifier les démonstrations d'une théorie donnée (cf. (Péter 1977)). Il est ainsi plus facile de démontrer que $a.n + b$ prend une infinité de valeurs premières si a et b sont premiers entre eux en utilisant des méthodes analytiques (théorie « idéale » des fonctions holomorphes) qu'en passant par une méthode « intuitive » n'utilisant que des résultats sur les nombres entiers. Il est même des cas où le recours à la théorie idéale est obligatoire, simplement parce qu'on ne connaît pas d'autres moyens de résoudre le problème. On peut également citer comme exemple concluant de cette attitude la construction de la géométrie projective (théorie idéale) à partir de la géométrie euclidienne traditionnelle.

Si l'ensemble de la communauté scientifique ne fut pas convaincu, nombre de mathématiciens se lancèrent cependant dans la réalisation du programme de Hilbert[18] et les résultats qu'ils allaient obtenir devaient se révéler assez inattendus.

5.4 L'arithmétique formelle

L'idée originelle de Hilbert était de construire un système formel des mathématiques, ou tout au moins de l'analyse. Mais le problème se révéla tellement complexe que lui et ses

18 Notons que ce programme ne se résumait pas à la démonstration de la consistance de l'arithmétique formelle. Il comprenait 23 points, dont le premier était précisément l'étude de l'hypothèse du continu de Cantor. Certains points du programme de Hilbert ne sont pas encore résolus aujourd'hui.

élèves se limitèrent à construire un système formel de l'arithmétique. Nous allons décrire ce système. Nous insistons cependant sur le fait que les définitions données ci-dessous n'ont pas toujours toute la précision nécessaire, mais la mise en place parfaitement rigoureuse d'un système formel de l'arithmétique dépasserait, et de beaucoup, le cadre de cet ouvrage.

L'arithmétique formelle est une extension de la théorie de la démonstration[19] du calcul des prédicats.

Définition 5.1 – Symboles – *Les symboles de l'arithmétique formelle seront :*

$$\leftrightarrow, \rightarrow, \wedge, \vee, \neg, \forall, \exists, =, +, ., ', 0, a, b, c, \ldots, |, (,)$$

Les lettres $a, b, c, \ldots$ sont les variables de notre langage. Afin d'en disposer d'une infinité, nous admettons également comme variable les termes de la forme $a_{||\ldots|}$[20]. Nous allons définir les termes de notre langage.

Définition 5.2 – Termes –

1. *0 est un terme ;*
2. *les variables sont des termes ;*
3. *si r et s sont des termes alors $(r)'$, $(r) + (s)$, $(r).(s)$ sont des termes.*

Tout terme est obtenu en appliquant un nombre fini de fois les règles précédentes.

Il nous reste à définir les formules de notre langage :

Définition 5.3 – Formules –

1. *si r et s sont des termes alors $(r = s)$ est une formule ;*
2. *si A et B sont des formules alors $(A \leftrightarrow B)$, $(A \rightarrow B)$, $(A \wedge B)$, $(A \vee B)$ et $(\neg A)$ sont des formules ;*
3. *si A est une formule et x une variable alors $(\forall x\, A)$ et $(\exists x\, A)$ sont des formules.*

Toute formule est obtenue en appliquant un nombre fini de fois les règles précédentes.

Munis du langage, nous allons pouvoir définir les axiomes et règles d'inférence du système formel $\mathbb{N}$.

Définition 5.4 – Axiomes et règles d'inférence standards – *Les axiomes et les règles d'inférence du calcul des prédicats s'appliquent dans le système formel $\mathbb{N}$.*

Les axiomes que nous allons définir ensuite sont appelés *axiomes non-logiques*. Pourquoi ce nom ? Les axiomes que nous avons définis jusqu'ici ne remettent pas en cause les notions du calcul des prédicats et en particulier la notion de conséquence valide. En revanche, les axiomes que nous allons introduire ne peuvent plus s'interpréter au niveau de la notion de conséquence valide que comme des hypothèses de calcul, dans la mesure où ils n'ont pas de vérité logique. C'est en raison de ces axiomes que les résultats du calcul des prédicats ne peuvent s'étendre directement à l'arithmétique formelle. Nous essaierons de donner de chacun de ces axiomes une interprétation matérielle intuitive.

---

19 Une théorie formelle est une théorie de la démonstration et non une théorie des modèles.

20 Ainsi $a_{||||}$ représente formellement ce que l'on note généralement a_4.

Axiome 5.1 – Récurrence –

$$A(0) \wedge \forall x \, (A(x) \rightarrow A(x')) \rightarrow A(x)$$

Si nous interprétons l'opérateur $'$ comme *successeur de*, alors l'axiome précédent se comprend immédiatement comme la définition de la récurrence : si la propriété est vraie pour 0 et que, si elle est vraie pour n, elle est vraie pour $n + 1$, alors elle est vraie pour tout n[21].

Axiome 5.2 – Quatrième axiome de Peano –

$$a' = b' \rightarrow a = b$$

Cet axiome formalise que : $a + 1 = b + 1 \rightarrow a = b$

Axiome 5.3 – Cinquième axiome de Peano –

$$\neg a' = 0$$

Il n'existe aucun nombre ayant 0 pour successeur.

Axiome 5.4 – Pseudo-Transitivité de l'égalité –

$$a = b \rightarrow (a = c \rightarrow b = c)$$

Axiome 5.5 – Égalité –

$$a = b \rightarrow a' = b'$$

Axiome fondateur de l'égalité et de la notion de successeur avec le quatrième axiome de Peano ; si $a = b$ alors $a + 1 = b + 1$.

Axiome 5.6 – 0 élément neutre pour l'addition –

$$a + 0 = a$$

Axiome 5.7 – Successeur et addition –

$$a + b' = (a + b)'$$

---

21 Notons qu'ici les variables sont prises dans le sens de la généralité, comme dans toute la suite du jeu d'axiomes. On pourrait donc tout aussi bien remplacer chaque axiome par sa clôture universelle.

Axiome 5.8 – 0 élément absorbant de la multiplication –

$$a.0 = 0$$

Définit 0 comme élément absorbant de la multiplication.

Axiome 5.9 – Distributivité de la multiplication –

$$a.b' = a.b + a$$

À partir des définitions données ci-dessus, il est possible de définir la suite des nombres 1,2,3... récursivement en posant $1 = 0'$, $2 = 1' = 0\ >>\ldots$

On ne peut définir dans $\mathbb{N}$ que des fonctions polynomiales, en raison du symbolisme extrêmement pauvre du langage. Mais l'on peut, en revanche, définir pour toute fonction arithmétique $f(x_1, \ldots, x_n)$ un prédicat $p(x_1, \ldots, x_n, y)$ qui est vrai si et seulement si $f(x_1, \ldots, x_n) = y$. On peut alors « parler » d'une fonction et opérer sur elle en travaillant sur le prédicat associé qui, lui, appartient bien au système formel. Ainsi considérons la fonction indicatrice[22] des nombres pairs $P(x)$ et son prédicat associé $p(x, y)$. Le théorème $P(x+2) = P(x)$ s'exprimera dans $\mathbb{N}$ par $\exists u\, \exists v\, (p(x+2, u) \wedge p(x, v) \wedge (u = v))$.

Il est également possible de définir les symboles traditionnels de l'arithmétique classique comme synonymes de formules plus complexes. C'est le cas de $>$ ou de $<$ ou de l'exponentiation. En fait, l'essentiel des relations familières au mathématicien sont définissables.

On pense donc que le système formel $\mathbb{N}$ que nous venons de définir a bien comme interprétation standard, l'arithmétique telle que nous la pratiquons couramment. Si tel n'était pas le cas, il est clair que notre formalisme n'atteindrait pas son but.

Nous ne donnerons pas d'exemple de démonstration formelle dans $\mathbb{N}$. De telles démonstrations sont excessivement lourdes. La démonstration du théorème $a = a$ ne prend pas moins de 17 lignes ! Nous utiliserons d'ailleurs tout au long des paragraphes suivants des démonstrations informelles.

5.5 Décidabilité, complétude et consistance

Nous devons bien nous rappeler le but d'Hilbert : il s'agit, maintenant que les bases formelles sont posées, de démontrer la consistance formelle de $\mathbb{N}$ (*i.e.*, il n'existe pas dans $\mathbb{N}$ une formule E telle que $\vdash E$ et $\vdash \neg E$).

En 1925, Ackermann publia une démonstration de consistance de $\mathbb{N}$. Mais deux ans plus tard, Von Neumann montrait que la démonstration d'Ackermann ne s'appliquait qu'à un sous-système de $\mathbb{N}$, dans lequel l'axiome 5.1 est remplacé par un axiome voisin, mais plus faible.

En fait, les tentatives de démonstration de la consistance étaient vouées à l'échec comme allait le montrer Gödel. La démonstration originale de Gödel n'est pas celle que nous allons donner. Nous présenterons la démonstration donnée par Kleene en 1943, démonstration qui s'appuie sur les machines de Turing.

---

22 La fonction f indicatrice d'un ensemble E est la fonction définie par : $\forall x \in E, f(x) = 1$ et $\forall x \notin E, f(x) = 0$.

5.5.1 Décidabilité et calculabilité

Nous avons déjà effleuré les problèmes de calculabilité et de décidabilité dans le chapitre consacré aux machines de Turing.

Rappelons la définition que nous avons donnée de la calculabilité :

Définition 5.5 – Calculabilité – *Une fonction est dite calculable s'il existe une procédure algorithmique finie permettant de calculer en un nombre de pas fini sa valeur pour tout argument de son domaine.*

La notion de décidabilité est très proche de la définition de calculabilité, mais s'applique à des problèmes de décision *i.e.*, à des classes de questions auxquelles la réponse est *oui* ou *non.* :

Définition 5.6 – Décidabilité – *Une classe de questions est dite décidable s'il existe une procédure algorithmique finie permettant de résoudre n'importe laquelle des questions de cette classe en un nombre fini de pas.*

Des problèmes classiques de décision dans un système formel sont : « Une suite de symboles donnée est-elle une formule bien formée ? » ou « Une suite d'expressions donnée est-elle la démonstration d'une formule ? » ou encore « Une formule donnée est-elle démontrable ? ».

Les deux premières questions sont décidables. Il n'en est pas de même pour la troisième. Nous connaissons des cas où l'on sait résoudre le problème de décision : il s'agit par exemple du calcul propositionnel, par application de la méthode des tables de vérité. En revanche, nous ne savons pas a priori s'il existe une procédure de décision pour $\mathbb{N}$. Or une telle procédure permettrait de résoudre de façon simple et mécanique tous les problèmes concernant l'arithmétique élémentaire non encore résolus aujourd'hui. On pourrait ainsi prouver que la conjecture de Goldbach[23] ou le grand « théorème » de Fermat[24] sont démontrables ou indémontrables en appliquant mécaniquement cet algorithme. Il semble difficile de trouver une telle procédure. Il semblait encore plus difficile de prouver qu'elle n'existait pas. C'est pourtant ce que Church a fait.

La thèse de Church-Turing, que nous avons présentée au chapitre 10, ramène la notion de calculabilité intuitive à la notion de *calculabilité* pour les machines de Turing. Nous avons déjà évoqué les arguments en faveur de cette thèse. Nous la supposons admise dans la suite de l'énoncé. Rappelons la notion de *décidabilité* au sens de Turing : un prédicat est décidable au sens de Turing s'il existe une machine de Turing capable de déterminer sa vérité ou sa fausseté. La thèse de Church-Turing implique que la décidabilité intuitive et la décidabilité au sens de Turing sont équivalentes.

5.5.2 Construction d'une fonction non-calculable

Considérons de nouveau les machines de Turing que nous avons présentées au chapitre 4. Une machine de Turing est entièrement déterminée par son tableau, qui représente son programme. Ainsi le tableau 5.2 détermine totalement une machine de Turing

---

23 Tout nombre pair est somme de deux nombres premiers.

24 $x^n + y^n = z^n$ ne possède pas de solution entière pour $n > 2$. Andrew Wiles, de l'Université de Princeton, a proposé, en Juin 1993, une démonstration de cette conjecture.

	B	0	1
z_0	(z_1, B, G)	$(z_0, 0, D)$	$(z_0, 1, D)$
z_1	$(z_h, 1, I)$	$(z_h, 1, I)$	$(z_1, 0, G)$

Table 5.2 – Programme d'une machine de Turing

(celle qui calcule la fonction $f(x) = x + 1$). Nous pouvons réécrire le programme précédent, en représentant un état z_i par i, sous la forme :

$$1, B, G, 0, 0, D, 0, 1, D; h, 1, I, h, 1, I, 1, 0, G$$

Tout programme de machine de Turing peut donc s'écrire à l'aide des dix-sept symboles suivants[25] :

$$h\ 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ B\ D\ G\ I\ ,\ ;$$

Tout programme de machine de Turing peut donc être écrit sous la forme d'un nombre en base 17. De même, tout mot de Γ^* (de cardinal 11) peut se traduire de façon unique en un élément de $\mathbb{N}$.

Définition 5.7 – Index d'une machine de Turing – *Le nombre i (en base 17) qui représente une machine de Turing est appelé* index *de la machine de Turing, machine que nous noterons $\mathcal{M}_i$*[26].

Il découle immédiatement des résultats précédents le théorème :

Théorème 5.1 – *L'ensemble des machines de Turing est dénombrable.*

Nous devons commencer à sentir le résultat plus proche. En effet, nous avons démontré au début de ce chapitre que l'ensemble des fonctions à variable entière était non-dénombrable, et nous venons de montrer que l'ensemble des fonctions que nous pouvons calculer est « seulement » dénombrable. Nous sommes donc déjà convaincus qu'il existe bien des fonctions non-calculables. Il nous suffit maintenant d'en exhiber une. Il nous reste quelques étapes à franchir avant d'en arriver là.

Définition 5.8 – *Nous posons $T(i, a, n)$ le prédicat T qui prend la valeur vraie lorsque la machine de Turing dont l'index est i, appliquée à l'argument a, calcule au bout de n étapes exactement un résultat que nous noterons $\varphi_i(a)$.*

Remarquons que $T(i, a, n)$ est également faux si i n'est pas l'index d'une machine de Turing valide. Nous allons établir deux résultats fondamentaux sur T et φ.

Théorème 5.2 – *Le prédicat $T(i, a, n)$ est décidable (au sens intuitif et au sens de Turing).*

Les valeurs de i, a et n étant données, il est possible de vérifier que i est bien l'index d'une machine de Turing valide. Si ce n'est pas le cas, T est faux. Si c'est le cas, nous

25 Si on utilise un alphabet décimal. Nous pourrions nous contenter d'un système de numération binaire pour numéroter les états. Dans ce cas, neuf symboles suffiraient.

26 Notons qu'il existe des index qui ne désignent pas des machines de Turing valides. Cela est sans importance pour la suite.

pouvons appliquer la machine de Turing $\mathcal{M}_i$ à a et regarder si au bout de n étapes la machine vient juste de terminer le calcul d'une valeur. Si c'est le cas $T(i, a, n)$ est vrai, sinon il est faux.

T étant décidable au sens intuitif, il est décidable au sens de Turing, d'après la thèse de Church-Turing.

Théorème 5.3 – *La fonction partielle φ_i est calculable.*

Nous disons que φ_i est une fonction *partielle* car elle n'est définie que pour les valeurs de a et de i telles qu'il existe un n tel que $T(i, a, n)$ est vrai. Dire que φ_i est une fonction partielle calculable consiste simplement à dire que *si* φ_i est définie pour l'index i et l'argument a, *alors* elle est calculable. Le résultat dans ces conditions est évident. Il suffit d'appliquer la machine $\mathcal{M}_i$ à l'argument a et attendre qu'elle ait calculé un résultat (ce qui arrivera car $\exists n\, T(i, a, n)$ est vrai).

Il est temps pour nous d'arriver au résultat fondamental de cette section :

Théorème 5.4 – *Soit la fonction ψ définie par :*

$$\psi(a) = \begin{cases} \varphi_a(a) + 1 & \textit{s'il existe } n \textit{ tel que } T(a, a, n) \\ 0 & \textit{sinon} \end{cases}$$

ψ n'est pas calculable.

Supposons en effet que ψ soit calculable. Alors il existe une machine de Turing $\mathcal{M}_j$ qui calcule ψ. Nous avons donc pour tout a : $\psi(a) = \varphi_j(a)$ et en particulier pour $a = j$ nous avons :

$$\psi(j) = \varphi_j(j)$$

Comme $\mathcal{M}_j$ calcule ψ, nous avons pour tout a, $\exists n\, T(j, a, n)$ En particulier, pour $a = j$, nous savons qu'il existe n tel que $T(j, j, n)$. Or en appliquant directement la définition de ψ nous obtenons :

$$\psi(j) = \varphi_j(j) + 1$$

Ce qui est en contradiction avec l'équation précédente. Q.E.D.

On remarquera que cette démonstration s'appuie une fois de plus sur la méthode de la diagonale cantorienne.

Il existe un corollaire important à ce théorème :

Théorème 5.5 – *Le prédicat $\exists n\, T(a, a, n)$ est indécidable.*

Si ce prédicat était décidable, nous pourrions pour un a donné décider s'il existe n tel que $T(a, a, n)$. Si tel n'était pas le cas nous saurions que la valeur de $\psi(a)$ est 0. Sinon, nous pourrions appliquer la machine de Turing $\mathcal{M}_a$ à l'argument a, récupérer le résultat du calcul au bout des n étapes, lui ajouter 1, ce qui nous donnerait la valeur de $\psi(a)$. Nous venons de décrire là une procédure[27] de calcul de ψ. Or ψ n'est pas calculable, donc, par réduction à l'absurde, $\exists n\, T(a, a, n)$ est indécidable. Plus généralement, le prédicat $\exists n\, T(i, a, n)$ est indécidable. On en conclut que le problème de l'arrêt d'une machine de Turing est, lui aussi, indécidable.

---

27 Il faut tout de même ajouter une précision : la procédure décrite ci-dessus n'est correcte que lorsque l'on sait *construire* par une machine de Turing la machine de Turing calculant φ_a à partir de l'index a (ce qui est le cas au vu de notre définition de la fonction calculant l'index : celle ci est clairement inversible). Dans le cas contraire, il faudrait inclure dans le programme de la machine calculant ψ tous les programmes de toutes les machines calculant tous les φ_a, ce qui est impossible, une machine de Turing devant avoir un nombre fini d'instructions.

5.5.3 Indécidabilité de $\mathbb{N}$

La démonstration de l'indécidabilité de $\mathbb{N}$ va s'appuyer sur le résultat précédent. Il semble clair que le prédicat $\exists n\, T(a, a, n)$ peut se formaliser dans $\mathbb{N}$. Les machines de Turing ne comprennent dans leur définition que des opérations arithmétiques élémentaires et le prédicat T ne fait appel à aucune notion autre qu'arithmétique ou liée au calcul des prédicats. Il doit donc exister dans $\mathbb{N}$ une formule qui a comme interprétation[28] $\exists n\, T(a, a, n)$. Appelons C_a cette formule.

Remarquons maintenant que :

$$\exists n\, T(a, a, n) \text{ entraîne } \{\vdash C_a \text{ dans } \mathbb{N}\}$$

En effet, si l'on sait qu'il existe un n tel que $T(a, a, n)$, on peut le démontrer informellement en construisant chacune des étapes de la machine de Turing associée. Une telle démonstration peut évidemment se formaliser dans $\mathbb{N}$.

Si nous supposons que seules les formules vraies sont démontrables[29], nous avons :

$$\{\vdash C_a \text{ dans } \mathbb{N}\} \text{ entraîne } \exists n\, T(a, a, n)$$

En fait, C_a représente exactement $\exists n\, T(a, a, n)$.

Théorème 5.6 – Indécidabilité de $\mathbb{N}$ – $\mathbb{N}$ *est indécidable* i.e., *il n'existe pas de procédure de décision permettant d'établir la démontrabilité d'une formule.*

En effet, si une telle procédure existait, il serait possible, pour a donné, de savoir si C_a est démontrable ou non. On disposerait donc d'une procédure de décision pour $T(a, a, n)$, ce qui est impossible.

Il n'existe donc aucune procédure algorithmique permettant de déterminer la vérité ou la fausseté d'une formule quelconque de $\mathbb{N}$ comme nous pouvions le faire avec les tables de vérité en calcul propositionnel.

5.5.4 Incomplétude de $\mathbb{N}$

Nous allons avoir besoin de quelques propriétés supplémentaires pour démontrer l'incomplétude de $\mathbb{N}$ (rappelons qu'il s'agit de montrer qu'il existe une formule F telle que l'on ait ni $\vdash F$, ni $\vdash \neg F$).

Notons tout d'abord que comme dans $\mathbb{N}$ seules les formules vraies sont démontrables, nous avons :

$$\{\vdash \neg C_a \text{ dans } \mathbb{N}\} \text{ entraîne } \neg\exists n\, T(a, a, n)$$

---

28 Ici se trouve un des points principaux de la démonstration. C'est parce qu'on parle ici d'interprétation qu'on parlera ensuite de vérité de C_a si $\exists n\, T(a, a, n)$ est vraie. Il doit y avoir adéquation de $\mathbb{N}$ avec l'interprétation que l'on en fait : si $\exists n\, T(a, a, n)$ est vraie, alors C_a est vraie sans qu'il soit encore question de savoir si elle est démontrable : elle est vraie parce que nous supposons que notre modèle est adéquat. Il faut faire bien attention de ne pas confondre *vrai* et *démontrable.*

29 Nous ne pouvons que le supposer, une telle propriété est, comme nous allons le voir, indémontrable par des méthodes métamathématiques. On peut cependant penser qu'il s'agit d'une condition indispensable pour la consistance de notre système formel. D'autre part, on peut remarquer que dans l'interprétation classique de l'arithmétique, les axiomes de $\mathbb{N}$ sont vrais et l'application d'une règle d'inférence sur des axiomes ou des formules vraies donne systématiquement une formule vraie. Ce qui précède n'est cependant pas une démonstration acceptable car elle fait appel à des méthodes non finitistes, puisqu'elle considère l'ensemble achevé des démonstrations, qui est infini.

Nous allons maintenant nous appuyer sur deux éléments fondamentaux liés à l'architecture de $\mathbb{N}$. La première est que dans $\mathbb{N}$, on peut reconnaître la démonstration d'une formule comme telle : la reconnaissance d'une démonstration est un problème décidable, comme nous l'avions déjà fait remarquer. Ensuite, nous devons remarquer que les symboles permettant de construire les formules de $\mathbb{N}$ étant en nombre fini, le nombre de formules et de démonstrations est donc dénombrable, et l'on peut en établir une énumération. Ce dernier point est la clé de voûte de l'ensemble des démonstrations d'incomplétude dans les systèmes formels ; toute démonstration d'incomplétude revient finalement à montrer qu'il y a un nombre non dénombrable de formules vraies, alors qu'il y a seulement un nombre dénombrable de démonstrations (de la même façon que l'on montre qu'il y a une infinité non-dénombrable de fonctions et seulement un nombre dénombrable de fonctions calculables).

Les deux points précédents étant établis, il devient évident qu'il est possible de construire une machine de Turing $\mathcal{M}_j$ qui, parcourant l'ensemble des démonstrations de $\mathbb{N}$, imprime 1 et s'arrête si elle trouve une démonstration de $\neg C_a$, et ne s'arrête jamais sinon. Ceci se résume par :

$$\exists n\, T(j,a,n) \text{ est équivalent à } \{\vdash \neg C_a \text{ dans } \mathbb{N}\ \}$$

Nous pouvons donc résumer les trois points que nous avons établis et qui vont nous permettre de poursuivre la démonstration :

1. $\{\vdash C_a$ dans $\mathbb{N}\ \}$ entraîne $\exists n\, T(a,a,n)$
2. $\{\vdash \neg C_a$ dans $\mathbb{N}\ \}$ entraîne $\neg\exists n\, T(a,a,n)$
3. $\exists n\, T(j,a,n)$ est équivalent à $\{\vdash \neg C_a$ dans $\mathbb{N}\ \}$

Nous pouvons maintenant démontrer le théorème d'incomplétude de Gödel. Supposons qu'il existe n tel que $T(j,j,n)$. Alors d'après (3), nous avons : $\vdash \neg C_j$. Mais alors (2) nous donne : $\neg\exists n\, T(j,j,n)$. Ceci contredit l'hypothèse. Donc nous avons $\neg\exists n\, T(j,j,n)$, donc $\neg C_j$ est vrai.

Mais d'autre part, $\neg\exists n\, T(j,j,n)$ impose d'après (3) que *non* $\vdash \neg C_j$.

Enfin, supposons que $\vdash C_j$; alors d'après (1), nous savons qu'il existe un n tel que $T(j,j,n)$, ce qui impliquerait que C_j soit vrai. Or ceci est contradictoire avec le premier point que nous avons démontré. Donc, on n'a pas $\vdash C_j$.

Nous pouvons résumer nos trois résultats dans le théorème suivant :

Théorème 5.7 – Théorème d'incomplétude – *Dans* $\mathbb{N}$, *nous pouvons trouver une formule liée* C_j *telle que*

1. *nous n'avons pas* $\vdash C_j$;
2. *nous n'avons pas* $\vdash \neg C_j$;
3. $\neg C_j$ *est vraie.*

Nous avons donc démontré que pourvu que $\mathbb{N}$ soit adéquat et que seules les formules vraies soient démontrables dans $\mathbb{N}$, $\mathbb{N}$ est incomplet, c'est-à-dire qu'il existe des formules (vraies d'après l'interprétation standard) qui ne sont pas démontrables et dont la négation n'est pas non plus démontrable.

Essayons de comprendre comment nous sommes arrivés à ce résultat ; reprenons (3)

$$\exists n\, T(j,j,n) \text{ est équivalent à } \{\vdash \neg C_j \text{ dans } \mathbb{N}\ \}$$

Dans l'interprétation que nous utilisons, $\neg C_j$ exprime $\neg\exists n\, T(j, j, n)$. Nous avons donc construit une formule qui dans l'interprétation considérée exprime sa propre indémontrabilité. Il s'agit d'une assertion très proche du paradoxe du menteur.

Le résultat qui découle du théorème d'incomplétude est qu'il est impossible de formaliser l'ensemble des propriétés de l'arithmétique standard dans un système formel. La formule $\neg C_j$ est une propriété de l'arithmétique, vraie, mais indémontrable. Une partie du programme de Hilbert disparaissait.

5.5.5 Non démontrabilité de la consistance de $\mathbb{N}$

Nous ne donnerons qu'une idée de la démonstration montrant qu'il est impossible de démontrer la consistance formelle de $\mathbb{N}$ dans le système formel $\mathbb{N}$.

Constatons tout d'abord que dans notre démonstration nous avons utilisé comme hypothèse : $\{ \vdash \neg C_a$ dans $\mathbb{N} \}$ entraîne $\neg\exists n\, T(a, a, n)$

En fait cette hypothèse peut se réduire à la notion de consistance de $\mathbb{N}$: il n'existe pas dans $\mathbb{N}$ de formule F telle que $\vdash F$ et $\vdash \neg F$. Il suffit pour s'en convaincre de constater que ($\{ \vdash \neg C_a$ dans $\mathbb{N} \}$ entraîne $\neg\exists n\, T(a, a, n)$) se démontre à partir de ($\exists n\, T(a, a, n)$ entraîne $\{ \vdash C_a$ dans $\mathbb{N} \}$) et de la consistance simple.

Donc nous avons : ($\mathbb{N}$ est simplement consistant) entraîne ($\neg C_j$ est vrai) (ici j représente le j particulier du théorème précédent).

La suite de la démonstration consiste à remarquer que l'on peut exprimer de façon formelle la proposition : $\mathbb{N}$ est simplement consistant. Il suffit de considérer que l'on peut construire une machine de Turing $\mathcal{M}_k$ cherchant dans les démonstrations de $\mathbb{N}$ une démonstration d'une formule de la forme $F \wedge \neg F$, qui imprime 1 si elle en trouve une et ne termine jamais sinon.

Alors la consistance de $\mathbb{N}$ est équivalente au fait que, pour un a quelconque, il n'existe pas de n tel que $T(k, a, n)$. En particulier, il ne doit pas exister de n tel $T(k, k, n)$. Ce prédicat peut être formalisé, nous le savons déjà, par une formule de $\mathbb{N}$ que nous appellerons C. Nous pouvons alors reprendre la proposition : ($\mathbb{N}$ est simplement consistant) entraîne ($\neg C_j$ est vrai). Elle devient formellement :

$$C \rightarrow \neg C_j$$

Hilbert et Bernays ont montré en 1939 que cette formule est démontrable dans $\mathbb{N}$, ce qui est assez naturel dans la mesure ou $\mathbb{N}$ doit être adéquat à notre modèle intuitif de l'arithmétique. Donc :

$$\vdash C \rightarrow \neg C_j$$

Supposons maintenant que $\vdash C$ (c'est-à-dire que la consistance de $\mathbb{N}$ est démontrable), alors nous obtenons $\vdash \neg C_j$, ce qui contredit le théorème précédent (nous avons montré par le théorème d'incomplétude que nous n'avons pas : $\vdash \neg C_j$).

D'où le résultat :

Théorème 5.8 – Indémontrabilité de la consistance de $\mathbb{N}$ – *La formule C exprimant formellement dans $\mathbb{N}$ la consistance de $\mathbb{N}$ est indémontrable dans $\mathbb{N}$.*

5.6 Les conséquences

Nous ne nous attarderons pas sur les conséquences du théorème de non-démontrabilité de la consistance. Disons simplement que, pour nombre de mathématiciens, il signifia la fin de l'espoir d'obtenir une garantie de la sûreté des mathématiques classiques.

En revanche, la notion de non-décidabilité et de non-calculabilité nous intéresse au premier chef. Ces résultats ont donné naissance à une branche fructueuse de l'informatique, la théorie de la calculabilité et de la complexité. Ils ont également fourni un argument aux défenseurs de l'IA : « dans la mesure où des fonctions ne sont pas calculables, et où nous voulons tout de même obtenir des résultats utilisables, nous sommes obligés d'utiliser des méthodes *heuristiques*, méthodes caractéristiques d'une approche IA. » Il faut cependant prendre ce raisonnement avec prudence ; un algorithme d'intelligence artificielle s'exécutera, comme tout autre algorithme, sur une machine de Turing, et est donc soumis aux mêmes limitations.

Le théorème d'incomplétude nous intéresse également dans la mesure où, selon certains, il montre qu'il existe des propositions vraies qu'aucun moyen mécanique ne permet de démontrer ; certains adversaires de l'IA se sont servis de cet argument pour prétendre qu'un calculateur ne pourrait jamais réaliser certains travaux réalisés par l'esprit humain. L'argument doit aussi être prudemment considéré.

On peut, d'autre part, se demander si la démonstration d'incomplétude ne vient pas de la trop grande pauvreté du jeu d'axiomes que nous avons utilisé au départ. Si nous ne parvenons pas à démontrer une propriété que nous savons vraie (comme celle que nous avons construite lors de la démonstration du théorème d'incomplétude), il nous suffit de l'inclure comme axiome supplémentaire. On pourrait ainsi espérer construire un système complet par ajout successif de toutes les propriétés vraies que nous ne pouvons démontrer. Il existe malheureusement une objection majeure à cette méthode. Une extension du théorème de Gödel stipule que pour tout système formel contenant l'arithmétique, il est possible de construire une propriété vraie non-démontrable, en utilisant d'ailleurs une méthode très voisine de celle que nous venons de présenter. Ceci ruine définitivement nos espoirs. Disons tout de même que le théorème d'incomplétude n'est pas aussi anti-naturel ou anti-intuitif qu'il peut apparaître au premier abord. En fait, il exprime que notre modèle intuitif de l'arithmétique est trop riche pour être réduit à un quelconque système formel, ce qui peut sembler relativement normal. Depuis la démonstration du théorème d'incomplétude, des modèles différents vérifiant l'axiomatique formelle de l'arithmétique ont été développés ; il a été démontré que l'ensemble de toutes les formules vraies dans tous les modèles est exactement l'ensemble des théorèmes.

Au sujet des modèles non-standard, Thoralf Skolem a démontré, en étendant un théorème dû à Lowenheim, que toute formalisation axiomatique du premier ordre de la théorie des ensembles (dont le modèle standard admet, bien entendu, des ensembles non-dénombrables) admet un modèle dont le domaine d'objets est au plus dénombrable. Donc la théorie naïve des ensembles ne peut être formalisée de façon catégorique[30] par un

---

30 Une théorie est catégorique ssi l'ensemble de ses modèles sont isomorphes *i.e.*, pour tout couple de modèles M_1, M_2, il existe une bijection f entre les domaines d'interprétation de ces deux modèles telle que pour tout symbole de prédicat P d'arité n, $M_1(P)(a_1, \ldots, a_n)$ est vrai ssi $M_2(P)(f(a_1), \ldots, f(a_n))$ est vrai et telle que pour tout symbole fonctionnel g d'arité n, $M_1(g)(a_1, \ldots, a_n)$ vaut a ssi $M_2(g)(f(a_1), \ldots, a_n)$ vaut $f(a)$. Tous les modèles d'une théorie catégorique sont donc mathématiquement indiscernables.

système axiomatique du premier ordre. Ce théorème n'est pas aussi paradoxal qu'il peut le paraître. En effet, toute théorie axiomatique formelle contient au plus, nous l'avons vu, un nombre dénombrable de démonstrations. Il semble clairement difficile dans ces conditions de représenter des ensembles non-dénombrables. On peut aussi mettre le théorème de Lowenheim-Skolem en relation avec le théorème d'incomplétude : considérons un système $\mathcal{S}$ « riche ». Comme tous les systèmes « riches » sont incomplets, on peut trouver une propriété p telle que les systèmes $\mathcal{S} \cup p$ et $\mathcal{S} \cup \neg p$ soient des systèmes « corrects ». Ces deux systèmes ne peuvent être catégoriques car les interprétations ne sont pas isomorphes.

Remarquons enfin la grande différence qui existe entre le théorème d'incomplétude et le théorème énonçant l'impossibilité de démontrer la consistance de $\mathbb{N}$ dans $\mathbb{N}$. Le premier stipule qu'il existe des propriétés vraies non démontrables. *Le second ne prétend en aucun cas qu'il existe des propriétés démontrables qui ne soient pas vraies.* Il dit simplement que l'on ne peut démontrer la consistance de $\mathbb{N}$ en utilisant seulement les axiomes de $\mathbb{N}$ et des méthodes finitistes, comme l'exigeait Hilbert. Une démonstration de consistance utilisant des méthodes non-finitistes a d'ailleurs été publiée par Gerhard Gentzen peu après que Gödel ait énoncé son théorème. Cette méthode satisfait de nombreux mathématiciens, même si elle fait appel à la notion d'induction transfinie. Il ne faut donc en aucun cas parler de théorème d'inconsistance comme on parle du théorème d'incomplétude, mais bien d'un théorème de non-démontrabilité *finitiste* de la consistance, ce qui est bien différent.

5.7 Faut-il jeter la logique ?

Les résultats obtenus par Gödel ont eu des conséquences très importantes sur la communauté des mathématiciens eux-mêmes. Certains d'entre eux, dont Imre Lakatos, ont même estimé nécessaire de renoncer à la théorie de la démonstration pour ne garder aux démonstrations mathématiques que leur caractère intuitif (Lakatos 1984) :

> « Pourquoi ne disons-nous pas que le test ultime permettant de savoir si une méthode peut-être admise en arithmétique revient à la question de savoir si elle est intuitivement convaincante ? »

Il faut cependant faire une distinction entre ce qui apparaît comme deux parties fondamentalement différentes de la logique, que l'on a parfois tendance à confondre :

- Il y a d'une part un problème à caractère philosophique qui s'interroge sur la valeur sémantique des axiomes logiques ; Est-ce qu'un axiome est vrai ? Qu'est-ce que la vérité ? Ce problème interpelle les philosophes anglo-saxons depuis près d'un siècle ; un bon livre faisant le point sur la philosophie de la logique est (Engel 1989).
- Il y a d'autre part un problème mathématique qui s'interroge sur la relation entre une théorie de la vérité, et une axiomatisation de cette théorie. C'est essentiellement à ce point que nous nous sommes intéressés.

En fait, ces deux points correspondent à deux étapes pragmatiques du travail du mathématicien ou du logicien. Face à une réalité trop vaste pour être traitée efficacement, il commence par la *réduire* en un langage formel dans lequel il définit une théorie de la vérité et un modèle, puis il construit une théorie de la démonstration qui lui permet

de vérifier la validité de ses intuitions dans le cadre de ce modèle. Les deux étapes sont indispensables, mais il ne faut pas attendre plus de la logique que ce qu'elle peut donner : sa démarche est réductionniste, et on ne peut trouver au bout de la chaîne un système aussi riche que le monde réel qu'elle modélise. La crise de la science actuelle est plus une crise liée aux limites de cette science qu'à ses fondements : ce qui est en cause, c'est le dogme *réductionniste* qui, si l'on caricature un peu, prétendait transformer *sans perte de généralité* le monde en un système axiomatique *fini* où tout serait démontrable. Il semble clair que le dogme a fait long feu, dans les deux domaines ; les philosophes semblent accepter l'impossibilité de réduire le langage ou la pensée humaine, ainsi que les notions intuitives de vérité et de validité sous-jacentes, à un système unique ; les mathématiciens sont conscients que la théorie sémantique et la théorie de la démonstration ne sont pas équivalentes, et qu'aucun système axiomatique fini ne peut représenter complètement et catégoriquement un modèle un peu complexe.

Il ne faut pourtant pas rejeter le travail accompli, ni refuser de le continuer. Il faut accepter la vision pragmatiste qui consiste à regarder la logique comme un outil, imparfait certes, mais pourtant indispensable. On ne peut rejeter les axiomes de la logique modale sous prétexte qu'elle est « née dans le péché » (Quine), si celle-ci fournit un modèle utile pour formaliser le raisonnement. De même, on ne peut rejeter la théorie de la démonstration, sous prétexte qu'elle ne fournit pas une certitude absolue : elle n'est qu'une vérification de nos intuitions et a rempli et continuera à remplir parfaitement ce rôle.

CHAPITRE 6

Calcul propositionnel et résolution

Jean-Marc Alliot — Thomas Schiex

6.1 Introduction

Nous nous sommes jusqu'à présent occupés de logique sur un plan purement mathématique. Le but de ce chapitre est de développer des méthodes algorithmiques permettant de déterminer la satisfiabilité d'une formule, d'un ensemble de clauses et de faire de la résolution automatique.

Nous nous intéresserons dans ce chapitre au calcul propositionnel, puis nous étendrons les résultats à la logique du premier ordre dans le chapitre suivant. Certains des résultats décrits dans ces deux chapitres ont conduit à la réalisation du langage PROLOG, devenu un langage classique de l'intelligence artificielle durant ces dernières années.

6.2 Principe de déduction

6.2.1 Résultats préliminaires

On a pu voir au chapitre 2 qu'une formule est dite insatisfiable ssi elle n'a pas de modèle. Cette notion s'étend simplement à un ensemble de formules :

Définition 6.1 – Ensemble de formules insatisfiable – *Un ensemble E de formules est insatisfiable, ou (sémantiquement) inconsistant, ssi il n'existe aucune interprétation M telle que chaque formule A de E soit satisfaite par M.*

En particulier, un ensemble de formules insatisfiable a pour conséquence logique toute formule, et en particulier la formule toujours fausse $\varnothing$. En fait, on déduit simplement de la définition de conséquence valide et d'insatisfiabilité qu'un ensemble de formules est insatisfiable ssi la formule $\varnothing$ en est une conséquence.

De façon analogue, il découle de la définition précédente et de la notion de conséquence valide que :

Théorème 6.1 – Principe de déduction – *Une formule A est la conséquence valide d'un ensemble E de formules ssi $E \cup \{\neg A\}$ est insatisfiable.*

En effet, toute interprétation est :

- soit un modèle de E, et c'est alors un modèle de A, et donc certainement pas un modèle de $E \cup \{\neg A\}$;
- soit ce n'est pas un modèle de E, et ce n'est donc pas, *a fortiori*, un modèle de $E \cup \{\neg A\}$.

Réciproquement, si $E \cup \{\neg A\}$ est insatisfiable, cela implique que toute interprétation qui satisfait E ne satisfait pas $\neg A$, et satisfait donc A.

6.2.2 Interprétation du résultat

L'intérêt du principe de déduction est qu'il ramène toute preuve de déduction à une preuve d'*inconsistance*, nous permettant dès à présent de nous concentrer uniquement sur les mécanismes de preuve de consistance (ou d'inconsistance) et de validité (une formule est valide ssi sa négation est inconsistante).

Nous connaissons déjà une méthode générale pour établir la validité ou la consistance (ou l'inconsistance) d'une formule, la méthode des tables de vérité. Cependant, cette méthode réclame, pour une formule contenant n atomes distincts, le calcul d'une table comprenant 2^n lignes. Il faut donc essayer de trouver des méthodes plus efficaces.

Il faut cependant signaler un résultat théorique important que nous reverrons au chapitre sur la complexité : le problème de la satisfiabilité d'une formule propositionnelle quelconque est $\mathsf{N}P$-complet. Sans rentrer pour l'instant dans les détails, cela signifie qu'il n'existe presque certainement aucun algorithme *efficace* permettant de calculer une affectation qui satisfait une formule, ou de vérifier sa validité.

Les deux algorithmes que nous présentons ci-après (algorithme de Quine et algorithme de réduction) sont notablement meilleurs dans certains cas précis que la méthode des tables de vérité, mais ne sont pas meilleurs en général. C'est ce qui nous conduira à nous intéresser ensuite aux formes clausales, aux clauses de Horn et au principe de résolution.

6.2.3 Les arbres sémantiques

La méthode des arbres sémantiques n'est pas plus efficace que la méthode classique des tables de vérité. Elle consiste à construire pour une formule F contenant les atomes de $S = \{p_1, \ldots, p_n\}$, un arbre binaire d'après les règles suivantes :

- chaque arc est étiqueté par un littéral p ou $\neg p$ avec $p \in S$. La branche p correspond à une affectation de t à p alors que la branche $\neg p$ correspond à l'affectation de f à p.
- les littéraux étiquetant les deux arcs issus d'un même nœud sont opposés.
- aucune branche ne comporte plus d'une occurrence de chaque atome.

Un arbre sémantique est dit *complet* si chaque branche contient une et une seule fois chaque atome. Il est dit *partiel* si chaque branche contient au plus une fois chaque atome. Il est clair qu'un arbre sémantique complet comprend 2^n feuilles. À chacune des feuilles, il faut réaliser une évaluation de la formule pour connaître sa validité. Il n'y a donc pas d'amélioration de la méthode des tables de vérité.

6.2.4 Algorithme de Quine

L'algorithme de Quine est une amélioration de la méthode des arbres sémantiques. Lors de la construction de l'arbre, on réalise à chaque nœud de l'arbre binaire une éva-

luation partielle de la formule, qui consiste à la simplifier en prenant en compte tous les atomes dont la valeur est déterminée. Si une évaluation partielle permet de conclure directement, on ne poursuit pas la construction de l'arbre à partir de ce nœud.

Voyons cela sur un exemple. Soit la formule $((p \rightarrow q) \wedge p) \rightarrow q$.

1. en donnant à p la valeur t, la formule est équivalente à $q \rightarrow q$, qui est valide. Nous ne poursuivrons donc pas la construction de l'arbre ;
2. en prenant pour p la valeur f, la formule est alors équivalente à $((\varnothing \rightarrow q) \wedge \varnothing) \rightarrow q$, qui est équivalent à $\varnothing \rightarrow q$, qui est valide puisque $\varnothing$ est toujours f. La construction de l'arbre est donc également terminée, et la formule est bien valide.

6.2.5 Algorithme de réduction

Cette méthode est basée sur un mécanisme de preuve par l'absurde. Elle peut être avantageusement utilisée quand la formule comprend de nombreuses implications. Elle consiste à supposer que la formule initiale est fausse, puis à en déduire les sous-valeurs logiques de chacune des deux sous-formules placées de chaque côté de l'implication et à itérer le processus jusqu'au bout.

Voici un exemple. Soit la formule : $((p \wedge q) \rightarrow r) \rightarrow (p \rightarrow (q \rightarrow r))$. On va supposer qu'il existe une assignation de p, q, r qui rend cette formule fausse. On en déduit alors :

1. $(p \wedge q) \rightarrow r$ est assignée à t ;
2. $(p \rightarrow (q \rightarrow r))$ est assignée à f.

À partir de ces éléments on peut appliquer à nouveau le processus sur (2) ; on obtient alors que p est t, et enfin que q est t et r est f. Or cette affectation est en contradiction avec (1). Donc la formule est valide.

6.3 Formes normales

Les méthodes que nous venons de voir sont inefficaces dans la pratique. Un certain nombre de méthodes, supposément plus efficaces en moyenne, tentent d'exploiter une forme simplifiée d'expression de formules.

6.3.1 Théorème de normalisation

Nous allons tout d'abord définir la notion de clause, puis celle de forme conjonctive normale.

Définition 6.2 – Clause – *Une clause est une disjonction de littéraux $p_1 \vee p_2 \ldots \vee p_n$, les littéraux pouvant être positifs ou négatifs.*

On emploie souvent pour noter une clause une notation ensembliste : une clause est un ensemble de littéraux. La clause vide $\varnothing$ est alors une formule toujours fausse (f est élément neutre pour la disjonction). Ceci justifie la notation $\varnothing$ employée jusqu'ici pour dénoter une formule toujours fausse.

Définition 6.3 – Forme conjonctive normale – *Une forme conjonctive normale est une conjonction de clauses :* $C_1 \wedge \ldots \wedge C_n$.

On considère parfois une forme conjonctive normale comme un ensemble de clauses et on la note alors : $\{C_1, \ldots, C_n\}$. On utilise ainsi souvent une terminologie ensembliste lorsque l'on parle de forme conjonctive normale (on dit ainsi qu'une clause *appartient* à une forme conjonctive normale...). Cette confusion est justifiée par le résultat classique : $\{H_1, \ldots, H_n\} \models A$ équivaut à $\models H_1 \wedge \ldots \wedge H_n \rightarrow A$.

Enfin, le résultat fondamental :

Théorème 6.2 – Théorème de normalisation – *Toute formule admet une forme conjonctive normale qui lui est logiquement équivalente.*

La preuve que nous allons donner de ce théorème sera une preuve constructive. À partir d'une formule quelconque, nous allons montrer comment construire la forme conjonctive normale associée.

1. on remplace toutes les occurrences de $(A \leftrightarrow B)$ par $(A \rightarrow B) \wedge (B \rightarrow A)$. Ceci supprime toutes les occurrences du connecteur $\leftrightarrow$;
2. on remplace toutes les occurrences de $(A \rightarrow B)$ par $(\neg A \vee B)$. Ceci supprime les occurrences de $\rightarrow$;
3. On applique récursivement les règles de réécriture suivantes :
   (a) $\neg(A \wedge B) \rightsquigarrow (\neg A \vee \neg B)$
   (b) $\neg(A \vee B) \rightsquigarrow (\neg A \wedge \neg B)$

   On ne trouve plus maintenant d'occurrence de $\neg$ que devant les atomes.
4. on supprime toutes les doubles occurrences de la négation : $\neg\neg A \rightsquigarrow A$. Il ne reste alors plus que des occurrences uniques de $\neg$ placées directement devant les atomes ;
5. on applique alors la règle de distributivité : $A \vee (B \wedge C) \rightsquigarrow (A \vee B) \wedge (A \vee C)$. (On peut également avoir à appliquer la commutativité de $\vee$ si l'expression est « dans le mauvais sens »).

La formule obtenue est une forme conjonctive normale, équivalente à la formule de départ.

6.3.2 Exemple

Nous allons examiner le fonctionnement de cette procédure sur un exemple. Considérons la formule :

$$p \leftrightarrow (q \rightarrow r)$$

1. on remplace d'abord l'équivalence pour obtenir : $(p \rightarrow (q \rightarrow r)) \wedge ((q \rightarrow r) \rightarrow p)$
2. on remplace ensuite les implications et l'on a : $(\neg p \vee (\neg q \vee r)) \wedge (\neg(\neg q \vee r) \vee p)$
3. on fait traverser à la négation les parenthèses : $(\neg p \vee \neg q \vee r) \wedge ((q \wedge \neg r) \vee p)$
4. on applique la distributivité de $\vee$: $(\neg p \vee \neg q \vee r) \wedge ((q \vee p) \wedge (\neg r \vee p))$
5. on regroupe les termes : $(\neg p \vee \neg q \vee r) \wedge (q \vee p) \wedge (\neg r \vee p)$

Le lecteur soupçonneux pourra construire les tables de vérité des deux formules pour vérifier qu'elles sont bien équivalentes.

6.3.3 Vocabulaire

Une clause d'une forme normale contenant deux littéraux opposés p et $\neg p$ est valide et peut être supprimée de la forme normale sans que son sens soit modifié.

Définition 6.4 – Forme conjonctive normale pure – *Une forme conjonctive normale ne contenant aucune clause valide est dite forme conjonctive normale pure.*

Quelques autres remarques utiles concernant les formes conjonctives normales peuvent être faites :

1. si une clause C_i, élément d'une forme normale, contient une clause C_j, élément de la même forme normale, la clause C_i peut être supprimée de la forme normale sans que son sens ne soit changé ;
2. une forme normale vide est valide ;
3. une forme normale contenant la clause vide est inconsistante ;
4. les formules valides ont une forme normale pure vide. *Le test de validité d'une forme normale est trivial* ;
5. on réduit classiquement deux clauses opposées $(A \wedge \neg A)$ en $\varnothing$, et n'importe quelle clause avec $\varnothing$, par exemple $A \wedge \varnothing$, à $\varnothing$. La clause $\varnothing$ est la seule clause inconsistante.

Le lecteur peut être surpris au premier abord par les points 2, 3 et 4. Qu'il se souvienne alors que $\{\varnothing\}$ est différent de $\varnothing$ et ces trois énoncés devraient s'éclairer.

6.3.4 Algorithme de Davis et Putnam

L'algorithme de Davis et Putnam (Davis and Putnam 1960) est un raffinement de l'algorithme de Quine appliqué aux formes normales conjonctives. Seul nous intéresse le problème de consistance, le test de validité étant dans ce cas trivial. Détaillons l'algorithme de Davis et Putnam. On considère une forme normale pure N.

1. si $N = \varnothing$ alors N est consistant et fin du calcul ; si N contient la clause vide $(\varnothing)$ alors N est inconsistant et fin du calcul. Sinon :
2. choisir un atome p tel que $p \in N$;
3. calculer les sous-ensembles de N :
   - N_p qui contient les clauses de N contenant p ;
   - $N_{\neg p}$ qui contient les clauses de N contenant $\neg p$;
   - N_c complémentaire de $N_p \cup N_{\neg p}$ dans N.
4. calculer $\mathcal{N}_p$ ensemble des clauses de N_p privées de p et $\mathcal{N}_{\neg p}$ ensemble des clauses de $N_{\neg p}$ privées de $\neg p$;
5. N est inconsistant ssi $\mathcal{N}_p \cup N_c$ et $\mathcal{N}_{\neg p} \cup N_c$ le sont.

Nous allons rapidement démontrer la validité de cet algorithme :

- si p est vrai, on peut alors supprimer le littéral $\neg p$ dans chaque clause le contenant puisqu'il est faux. D'autre part, toutes les clauses contenant p sont vraies et peuvent donc être supprimées. L'ensemble $N = N_p \cup N_{\neg p} \cup N_c$ se réduit alors $N = \mathcal{N}_{\neg p} \cup N_c$;
- si p est faux, on peut appliquer une opération symétrique et l'ensemble N se réduit alors à $N = \mathcal{N}_p \cup N_c$;

– pour que N soit inconsistant, il faut que les deux ensembles cités plus haut soient inconsistants puisqu'ils correspondent chacun à une des interprétations possibles de p, et qu'une formule est inconsistante si aucune interprétation ne permet de la rendre vraie.

Remarquons que cet algorithme réduit strictement à chaque étape le problème originel en deux problèmes plus simples dans lesquels ne figure plus un des atomes de la formule de départ. Il est donc clair que cet algorithme aboutit.

On peut aussi noter la parenté avec l'algorithme de Quine. Il s'agit là aussi de construire un arbre binaire, chaque sommet portant un arc correspondant à $\mathcal{N}_p \cup N_c$ (arc $\neg p$ chez Quine) et un arc correspondant $\mathcal{N}_{\neg p} \cup N_c$ (arc p chez Quine) pour tous les $p \in N$.

Deux règles heuristiques simples de choix de l'atome peuvent être précisées pour cet algorithme :

1. *Si N contient une clause contenant seulement p (respectivement $\neg p$), alors sélectionner p.* En effet, l'ensemble $\mathcal{N}_p$ (respectivement $\mathcal{N}_{\neg p}$) sera alors réduit à $\varnothing$, et l'ensemble $\mathcal{N}_p \cup N_c$ (respectivement $\mathcal{N}_{\neg p} \cup N_c$) sera alors inconsistant, ramenant la preuve d'inconsistance à un seul ensemble au lieu de deux.
2. *Si un seul des littéraux p et $\neg p$ apparaît dans N, alors sélectionner p.* Dans ce cas un des ensembles N_p ou $N_{\neg p}$ sera vide, ramenant de nouveau la preuve à l'inconsistance d'un seul ensemble.

On améliore ainsi considérablement l'efficacité de l'algorithme.

6.3.5 L'évaluation sémantique

Comme on vient de le voir, l'application de l'algorithme de Davis & Putnam à une formule N en forme normale conjonctive contenant n littéraux revient à la construction d'un arbre binaire dont les 2^n feuilles *potentielles* correspondent aux 2^n interprétations possibles des littéraux de N (un échec de la procédure de Davis & Putnam permet donc d'exhiber un modèle de la formule). Nous nous proposons d'étudier ici un algorithme récent étendant l'algorithme de Davis & Putnam : l'évaluation sémantique (Oxusoff and Rauzy 1989). Des améliorations sont introduites :

– via des *propriétés de coupure* qui permettront d'élaguer quelque peu l'arbre binaire des interprétations ;
– via l'utilisation d'*heuristiques*[1] destinées à guider le choix du prochain littéral p sélectionné.

Soient $p_1, \ldots, p_n$ les littéraux présents dans la formule N qui est sous forme normale conjonctive. Une interprétation partielle de N, ainsi que N, peuvent se représenter sous la forme d'une matrice creuse dont chaque ligne correspond à une clause de N et dont chaque colonne correspond à un littéral (positif et négatif).

L'ensemble des 5 clauses[2] suivantes $c_1, \ldots, c_5$:

---

1 Un ensemble d'heuristiques a été présenté pour la procédure de Davis & Putnam, de nouvelles heuristiques sont introduites dans l'évaluation sémantique.

2 Nous employons pour noter les clauses la notation dite « de Kowalski », dans laquelle une clause (disjonctions de littéraux positifs et négatifs) de la forme $a \vee b \vee c \vee \ldots \vee \neg x \vee \neg y \vee \neg z$ est notée $\ldots xyz \rightarrow abc \ldots$.

$$\begin{cases} c_1 : & \rightarrow A & =_{def} & A \\ c_2 : & A \rightarrow B, C & =_{def} & \neg A \vee B \vee C \\ c_3 : & A \rightarrow D, E & =_{def} & \neg A \vee D \vee E \\ c_4 : & D \rightarrow B & =_{def} & \neg D \vee B \\ c_5 : & E, C \rightarrow & =_{def} & \neg E \vee \neg C \end{cases}$$

se représenteront sous la forme de la matrice :

	A	$\neg A$	B	$\neg B$	C	$\neg C$	D	$\neg D$	E	$\neg E$
c_1 :	0									
c_2 :		0	0		0					
c_3 :		0					0		0	
c_4 :			0					0		
c_5 :						0				0

Par exemple, l'affectation de la valeur f à la proposition C dans une interprétation conduira à l'écriture d'un -1 dans toutes les cases (existantes) de la colonne correspondant à C et d'un 1 dans toutes les cases (existantes) de la colonne correspondant à $\neg C$.

Dans notre exemple, on aboutit à la matrice :

	A	$\neg A$	B	$\neg B$	C	$\neg C$	D	$\neg D$	E	$\neg E$
c_1 :	0									
c_2 :		0	0		-1					
c_3 :		0					0		0	
c_4 :			0					0		
c_5 :						1				0

Une interprétation sera un modèle de N dès lors que chaque ligne contient un 1 (chaque clause est alors trivialement vraie). Il est possible, à tout instant, de passer de la matrice à l'ensemble des clauses correspondant en supprimant toutes les clauses correspondant à une ligne contenant un 1 et tous les littéraux correspondant à une colonne qui contient un -1.

Élagage Les deux propriétés suivantes sont déjà utilisées dans la procédure de Davis & Putnam :

- dès que notre matrice contient une ligne vide (correspondant à la clause vide), toute interprétation construite en-dessous du nœud correspondant dans l'arbre binaire ne peut être un modèle. Il est donc inutile de parcourir le sous-arbre ayant ce nœud pour racine ;
- si la matrice n'a plus de ligne avant l'affectation de toutes les variables, on a déjà un modèle de N. Il est donc inutile de parcourir le sous-arbre correspondant à l'affectation des variables restantes.

À ces propriétés s'ajoute le théorème de partition des modèles :

Théorème 6.3 – *Soit G une formule en forme normale conjonctive, formant un ensemble de clauses, soit H un sous-ensemble de G. S'il existe un ensemble de littéraux positifs ou négatifs*

$L = \{l_1, \ldots, l_k\}$ tel que L est un modèle de H et tel qu'aucun des l_i ni des $\neg l_i$ n'apparaît dans $G \setminus H$, alors G est satisfiable si et seulement si $G \setminus H$ est satisfiable.

L'application de ce théorème est immédiate : soient N_i et N_j, deux nœuds de l'arbre binaire tels que N_j est dans la descendance de N_i, et G_i et G_j, les ensembles de clauses qui correspondent à ces deux nœuds. Pour passer de N_i à N_j, il a fallu affecter les littéraux $\{l_1, \ldots, l_k\}$. Si G_j est inclus dans G_i, alors (d'après le théorème) G_i est satisfiable si et seulement si G_j est satisfiable. De ce fait, si en développant le nœud N_j on s'aperçoit que G_j est inconsistant, on peut en déduire que G_i l'est aussi, et il devient inutile d'explorer les branches pendantes situées entre N_i et N_j.

Le test de l'inclusion d'un ensemble de clauses dans un autre est simplifié dans l'algorithme d'évaluation sémantique par l'utilisation des matrices : G_j, correspondant à un nœud N_j descendant de N_i, est inclus dans G_i si et seulement si toutes les lignes de la matrice qui comportent une case dans une des colonnes correspondant aux opposés des littéraux $\{l_1, \ldots, l_k\}$ affectés, comportent une autre case contenant un 1.

D'autre part, si G_j n'est pas inclus dans G_i, alors G_j n'est inclus dans aucun des ensembles de clauses associés aux nœuds ancêtres de N_i. Il est donc possible d'arrêter les tests d'inclusion en cas d'échec.

Considérons par exemple l'ensemble de 5 clauses suivantes :

$$\begin{cases} c_1 : & A \rightarrow B, C & =_{def} & \neg A \vee B \vee C \\ c_2 : & A, B, C \rightarrow & =_{def} & \neg A \vee \neg B \vee \neg C \\ c_3 : & C, D \rightarrow & =_{def} & \neg C \vee \neg D \\ c_4 : & \rightarrow A, D & =_{def} & A \vee D \\ c_5 : & D \rightarrow A & =_{def} & \neg D \vee A \end{cases}$$

On considère le parcours d'arbre suivant :

1. affectation de A à faux, nous amenant à un nœud N_1 où l'ensemble des clauses est $G_1 = \{\neg C \vee \neg D, D, \neg D\}$;
2. affectation de B à vrai, puis de C à faux, nous amenant à l'ensemble de clauses $G_2 = \{D, \neg D\}$, inconsistant, inclus dans G_1.

Il est alors inutile de reconsidérer les valeurs de vérité de C ou de B, on peut effectuer directement un retour arrière intelligent[3] sur le choix de A faux.

Fonction heuristique La fonction heuristique est chargée de choisir le littéral à affecter, et éventuellement la valeur à lui donner. L'heuristique utilisée (de nombreuses versions sont proposées dans (Oxusoff and Rauzy 1989)) s'appuie sur les principes suivants :

- choix des littéraux apparaissant dans une clause ne contenant qu'un littéral (mono-littéraux). Dans ce cas, le nœud correspondant dans l'arbre binaire n'aura qu'un fils. Par extension, choix d'un littéral dans les clauses les plus courtes (afin de provoquer l'apparition de clauses mono-littérales) ;

3 L'idée du théorème de partition est d'ailleurs très proche des idées du « backtrack intelligent » ou « backjumping » en PROLOG ou dans les CSP. L'ensemble de clauses G_2 est inclus dans G_1 parce que les différentes affectations réalisées entre G_1 et G_2 ont globalement supprimé des clauses (ou contraintes booléennes), elles ne peuvent donc pas être à l'origine de l'inconsistance et il est possible de revenir sur le choix ayant causé l'inconsistance (cf. (Gaschnig 1977; Dechter 1990; Bruynooghe and Pereira 1984; Schiex and Verfaillie 1993b)).

- choix de littéraux dont l'opposé n'apparaît pas dans l'ensemble des clauses (littéraux purs). Dans ce cas, le nœud correspondant dans l'arbre binaire n'aura qu'une branche à développer (cf. Davis & Putnam).

À ces deux critères de choix s'ajoute le choix des littéraux dans les plus jeunes clauses raccourcies (par suppression d'un littéral). Il sera ainsi possible de faire disparaître les clauses raccourcies « empêchant » l'application du théorème de partition.

Notons que depuis la fin de la décennie 1980, le problème de satisfaction de clauses (SAT) bénéficie d'un regain d'intérêt (Siegel 1987; Bryant 1986; Brace *et al.* 1990; Palmade 1992; Hooker 1988; Oxusoff and Rauzy 1989; Selman *et al.* 1992), probablement en liaison avec l'apparition des langages de programmation logique avec contraintes, des ATMS et avec la puissance des machines actuelles, permettant de traiter certains problèmes NP-complets (cf. chapitre 11) sur des tailles « réalistes ».

6.3.6 Révision rapide

Nous allons, sur un exemple, réviser quelques-uns des éléments vus jusqu'à présent dans ce chapitre.

Nous allons démontrer que : $\{p \rightarrow r, q \rightarrow r\} \models (p \vee q) \rightarrow r$.

1. tout d'abord, d'après le théorème de déduction, l'énoncé précédent se ramène à : $\{p \rightarrow r, q \rightarrow r, \neg((p \vee q) \rightarrow r)\} \models \varnothing$
2. il nous reste donc à démontrer : $\{p \rightarrow r, q \rightarrow r, \neg((p \vee q) \rightarrow r)\}$ est inconsistant.
3. en ramenant notre ensemble de clauses en forme normale, nous obtenons : $\{\neg p \vee r, \neg q \vee r, p \vee q, \neg r\}$
4. nous allons maintenant appliquer l'algorithme de Davis et Putnam. Nous choisissons r comme premier atome. Nous construisons alors les ensembles :
   - $N_r = \{\neg p \vee r, \neg q \vee r\}$, $N_{\neg r} = \{\neg r\}$ et $N_c = \{p \vee q\}$
   - $\mathcal{N}_r = \{\neg p, \neg q\}$ et $\mathcal{N}_{\neg r} = \varnothing$
   - $\mathcal{N}_r \cup N_c = \{\neg p, \neg q, p \vee q\}$ et $\mathcal{N}_{\neg r} \cup N_c = \{\varnothing, p \vee q\}$ qui est inconsistant car il contient $\varnothing$.
5. nous sommes ramenés à démontrer l'inconsistance de $N = \{\neg p, \neg q, p \vee q\}$. Nous choisissons comme atome p et appliquons de nouveau l'algorithme :
   - $N_p = \{p \vee q\}$, $N_{\neg p} = \{\neg p\}$ et $N_c = \{\neg q\}$.
   - $\mathcal{N}_p = \{q\}$, $\mathcal{N}_{\neg p} = \varnothing$
   - $\mathcal{N}_p \cup N_c = \{q, \neg q\}$ qui est inconsistant car q et $\neg q$ se réduit à $\varnothing$ et $\mathcal{N}_{\neg p} \cup N_c = \{\varnothing, \neg q\}$ qui est inconsistant car il contient $\varnothing$.
6. nous avons terminé notre démonstration d'inconsistance, ce qui prouve la correction de la formule initiale.

L'algorithme de Davis et Putnam s'est révélé particulièrement efficace dans le cas présent (alors que dans le cas général il n'est pas plus efficace que celui de Quine). Pourquoi ? En fait, c'est l'utilisation des deux heuristiques données à la section 6.3.4 qui permettent cette relative qualité.

6.4 Principe de résolution

Le principe de résolution est un système formé d'une unique règle d'inférence (il ne comprend aucun axiome). Sa grande simplicité de mise en œuvre en fait une méthode très utilisée.

6.4.1 Théorème de résolution

Nous avons vu qu'un ensemble de clauses est inconsistant ssi $\varnothing$ est une conséquence valide de cet ensemble. L'algorithme de résolution va construire des conséquences valides de notre ensemble N de départ jusqu'à l'obtention de la clause vide.

Théorème 6.4 – Principe de résolution – *Soit N une forme normale, C_1 et C_2 deux clauses de N. Soit p un atome tel que $p \in C_1$ et $\neg p \in C_2$. Soit la clause $R = C_1 \setminus \{p\} \cup C_2 \setminus \{\neg p\}$. Alors les formes normales N et $N \cup R$ sont logiquement équivalentes. La clause R est appelée* résolvante *de C_1 et C_2.*

La preuve de ce théorème est élémentaire ; elle repose sur la propriété suivante : si A, B et C sont trois formules, alors :

$$\{A \vee C, B \vee \neg C\} \models A \vee B$$

On retrouve immédiatement le principe de résolution en prenant pour C l'atome p.

Comme nous l'avons déjà dit, le principe de résolution définit un système d'inférence très simple, adéquat, mais non complet. Il est cependant *complet pour la réfutation* : une formule en forme normale est inconsistante ssi la clause $\varnothing$ peut être produite via application répétée du principe de résolution.

6.4.2 Application

Nous allons reprendre la formule : $\{p \rightarrow r, q \rightarrow r\} \models (p \vee q) \rightarrow r$. Nous avions été ramenés à démontrer l'inconsistance de : $\{\neg p \vee r, \neg q \vee r, p \vee q, \neg r\}$.

Nous allons appliquer le principe de résolution pour démontrer cette inconsistance ; nous allons numéroter nos clauses et dérouler les conséquences valides :

1. $\neg p \vee r$: hypothèse
2. $\neg q \vee r$: hypothèse
3. $p \vee q$: hypothèse
4. $\neg r$: hypothèse
5. $\neg p$: à partir de 1 et 4
6. $\neg q$: à partir de 2 et 4
7. q : à partir de 5 et 3
8. $\varnothing$: à partir de 6 et 7.

Le résultat a été obtenu de façon rapide, principalement parce que l'on a su choisir les bonnes clauses dans le bon ordre. Une technique « systématique » qui consisterait, par exemple, à choisir les atomes dans l'ordre lexicographique et les clauses dans l'ordre dans lequel elles se présentent, peut conduire à des temps de calcul prohibitifs, voire à des bouclages. Le principe de résolution ne fournit donc pas en général de méthode plus efficace que les algorithmes vus précédemment (cf. (Chang and Lee 1987) pour plus de détails).

6.5 Les clauses de Horn

6.5.1 Résultats généraux

Nous venons de voir que la méthode de résolution est, dans le cas général, inefficace. Il existe cependant des cas particuliers où cette méthode est rapide.

Nous allons nous intéresser dans ce chapitre à un type particulier de forme normale, les formes normales composées de *clauses de Horn.*

Définition 6.5 – Clauses de Horn – *Une clause de Horn est une clause comportant au plus un littéral positif. Il existe donc trois types de clauses de Horn :*

- *celles comportant un littéral positif et au moins un littéral négatif, appelées clauses de Horn* strictes*;*
- *celles comportant exactement un littéral positif et aucun littéral négatif, appelées clauses de Horn* positives*;*
- *celles ne comportant que des littéraux négatifs, appelées clauses de Horn* négatives *(et dont fait partie la clause vide* $\varnothing$*).*

6.5.2 Clauses de Horn et résolution

Nous reviendrons au paragraphe 6.5.4 sur l'interprétation intuitive que l'on peut donner des différents types de clauses de Horn. Nous allons tout d'abord nous intéresser à une amélioration de l'algorithme de résolution appliqué à une forme normale N ne contenant que des clauses de Horn :

1. si $\varnothing$ est dans N, l'ensemble est inconsistant et la résolution est terminée ;
2. sinon choisir une clause C et une clause P telles que P est une clause de Horn positive réduite à p et C une clause contenant $\neg p$;
3. calculer la résolvante R de P et C ;
4. Reprendre l'algorithme en remplaçant N par $(N \setminus \{C\}) \cup R$.

Nous nous permettons ici de remplacer N par $(N \setminus \{C\}) \cup R$ alors que l'algorithme général demande d'utiliser $N \cup R$. Mais dans le cas présent la clause C est une conséquence valide de R, et peut donc être omise. Nous engendrons donc bien des conséquences valides de N.

Cet algorithme peut se terminer de deux façons :

1. nous avons abouti à $\varnothing$. L'ensemble est alors inconsistant ;
2. il est impossible de poursuivre car nous ne parvenons plus à trouver P et C vérifiant les conditions requises. La formule est alors consistante et le modèle composé de chacune des clauses positives restantes (ou plus exactement le modèle composé de chacun des atomes de ces clauses) lorsque l'algorithme se termine, est un modèle de N[4].

On remarque immédiatement que la résolvante de P et C n'est autre que $C \setminus \{\neg p\}$. On supprime donc à chaque étape un atome dans une des clauses de N. On est donc

4 On appelle d'ailleurs ce modèle le *modèle canonique de N.*

d'autre part certain que l'algorithme aboutit. En ce qui concerne la complexité, il existe un algorithme linéaire (Dowling and Gallier 1984) résolvant le problème de la satisfiabilité d'un ensemble de clauses de Horn.

6.5.3 Exemple de résolution

Nous allons utiliser cette méthode pour démontrer l'inconsistance de :

$$\{\neg p \vee r, \neg r \vee s, p, \neg s\}$$

1. par sélection de p et $\neg p \vee r$, l'ensemble se réduit à : $\{r, \neg r \vee s, p, \neg s\}$
2. par sélection de r et $\neg r \vee s$, nous obtenons : $\{r, s, p, \neg s\}$
3. enfin par sélection de s : $\{r, s, p, \varnothing\}$

L'ensemble de clauses est bien inconsistant.

6.5.4 Interprétation intuitive

La question que nous devons maintenant nous poser est la suivante : quel est le cadre d'application des clauses de Horn, ou encore que peut-on représenter avec des clauses de Horn ?

- les clauses de Horn positives p sont appelées *faits*. Il s'agit en effet de l'énoncé de la vérité logique d'un atome ;
- les clauses de Horn strictes $q \vee \neg p_1 \vee \ldots \vee \neg p_n$ sont équivalentes à :

$$\{p_1, \ldots, p_n\} \models q$$

  Elles représentent des règles du type *si* ... *alors* .... Elles permettent de déduire de nouveaux faits à partir de faits existants ;
- les clauses négatives peuvent se comprendre comme des buts à atteindre. Considérons que nous souhaitions prouver $\{H_1, \ldots, H_n\} \models (p \wedge q \wedge r)$. La partie $(p \wedge q \wedge r)$ est le but de notre résolution. En appliquant une technique d'inconsistance nous sommes ramenés à $\{H_1, \ldots, H_n, (\neg p \vee \neg q \vee \neg r)\} \models \varnothing$. La clause $(\neg p \vee \neg q \vee \neg r)$ est une clause de Horn négative qui modélise donc bien le but à atteindre.

Ainsi, dans l'exemple précédent, l'ensemble $\{\neg p \vee r, \neg r \vee s, p, \neg s\}$ pouvait se lire : $\{p \rightarrow r, r \rightarrow s\} \models (p \rightarrow s)$, qui modélise la transitivité de l'implication.

Nous sommes maintenant en possession de presque tous les éléments nécessaires à l'introduction du langage PROLOG. Si nous ne souhaitions utiliser que des variables propositionnelles et non des prédicats, nous pourrions même nous en tenir là. Mais PROLOG opérant en logique du premier ordre, nous allons tout d'abord devoir étendre nos résultats au calcul des prédicats.

CHAPITRE 7

Calcul des prédicats et résolution

Jean-Marc Alliot — Thomas Schiex

7.1 Introduction

Notre but dans ce chapitre est d'étendre les résultats obtenus pour le calcul propositionnel au calcul des prédicats. Nous ne reviendrons pas sur certains résultats des chapitres précédents, supposés connus.

Ce chapitre présente les résultats fondamentaux qui ont conduit à la réalisation des systèmes de résolution de programmation en logique (PROLOG).

7.2 Formes normales

Nous allons ici définir la notion de forme prénexe pour une formule en calcul des prédicats.

7.2.1 Formes prénexes

La principale différence syntaxique entre calcul propositionnel et calcul des prédicats est la présence de quantificateurs à l'intérieur des formules. L'idée à la base de ce paragraphe est de construire une formule équivalente dans laquelle les quantificateurs seront rejetés en tête de la formule.

Définition 7.1 – Matrice – *Une matrice est une formule du calcul des prédicats ne contenant aucun quantificateur.*

Définition 7.2 – Forme prénexe – *Une forme prénexe est une formule du calcul des prédicats de la forme : $Q_1 x_1 \ldots Q_n x_n M$, où Q désigne $\exists$ ou $\forall$ et M est une matrice.*

Il nous reste à énoncer le résultat fondamental :

Théorème 7.1 – Forme prénexe équivalente – *Toute formule admet une forme prénexe équivalente.*

On peut donner une preuve constructive de ce théorème.

1. supprimer les connecteurs d'équivalence et d'implication (voir 6.3.1) ;

2. renommer certaines variables liées de manière à n'avoir plus de variable quantifiée deux fois en utilisant les équivalences entre $\mathcal{Q}x\,A(x)$ et $\mathcal{Q}y\,A(y)$ où $\mathcal{Q}$ est un quantificateur ;
3. supprimer les quantificateurs inutiles (dont la variable quantifiée n'apparaît pas dans leur portée) s'il y en a ;
4. transférer toutes les occurrences de la négation devant les atomes en utilisant les règles de réécriture vues au chapitre précédent (voir 6.3.1) et les règles supplémentaires suivantes :
   (a) $\neg\forall x A \rightsquigarrow \exists x \neg A$
   (b) $\neg\exists x A \rightsquigarrow \forall x \neg A$
5. faire passer les quantificateurs en tête en utilisant les règles de réécriture suivantes (et en utilisant l'associativité, la commutativité ou le renommage de variable si nécessaire !) :
   (a) $(\forall x A \wedge B) \rightsquigarrow \forall x(A \wedge B)$ si B ne contient pas x ;
   (b) $(\exists x A \wedge B) \rightsquigarrow \exists x(A \wedge B)$ si B ne contient pas x ;
   (c) $(\forall x A \vee B) \rightsquigarrow \forall x(A \vee B)$ si B ne contient pas x ;
   (d) $(\exists x A \vee B) \rightsquigarrow \exists x(A \vee B)$ si B ne contient pas x.

Il ne nous reste plus qu'à tester ce mécanisme sur un exemple.

7.2.2 Exemple

Nous allons construire la forme prénexe équivalente à la formule :

$$\forall x p(x) \wedge \exists y q(y) \rightarrow \exists y(p(y) \wedge q(y))$$

1. $\neg(\forall x p(x) \wedge \exists y q(y)) \vee \exists y(p(y) \wedge q(y))$ par suppression de $\rightarrow$;
2. $\neg(\forall x p(x) \wedge \exists y q(y)) \vee \exists z(p(z) \wedge q(z))$ par renommage des variables ;
3. $(\exists x \neg p(x) \vee \forall y \neg q(y)) \vee \exists z(p(z) \wedge q(z))$ par transfert de la négation ;
4. $\exists x \forall y \exists z(\neg p(x) \vee \neg q(y) \vee (p(z) \wedge q(z))$ par déplacement des quantificateurs.

Toute formule admettant une forme prénexe qui lui est logiquement équivalente, nous nous limiterons, à compter de ce paragraphe, aux formes prénexes. Pour des raisons de simplicité technique, nous considérerons également que toutes les formules manipulées sont *closes*.

7.3 Skolémisation

7.3.1 Formes de Skolem

L'idée qui se trouve derrière la construction des formes de Skolem est la suivante : les preuves de consistance sur les formules du calcul des prédicats sont compliquées, et il est difficile d'établir des méthodes générales de résolution. On va donc chercher à construire

une formule dont la consistance est équivalente à la consistance de la formule initiale, mais de forme plus simple, avec laquelle il sera plus aisé de travailler.

Il faut bien noter que la formule construite n'est pas logiquement équivalente à la formule initiale. Seules les consistances des deux formules sont équivalentes, ce qui nous suffit dans le cadre de preuve par inconsistance.

7.3.2 Algorithme de Skolémisation

Voici l'algorithme permettant de construire la forme de Skolem S_F d'une formule F :

- on transforme la formule F en une forme prénexe ;
- on remplace toute variable quantifiée existentiellement par un symbole de fonction dont les arguments sont les variables qui sont quantifiées universellement qui précèdent notre variable ;
- on supprime les quantificateurs existentiels, devenus inutiles.

7.3.3 Exemple

Nous allons reprendre l'exemple précédent :

$$\forall x p(x) \wedge \exists y q(y) \rightarrow \exists y (p(y) \wedge q(y))$$

1. nous savons que la transformation en forme prénexe nous donne :

$$\exists x \forall y \exists z (\neg p(x) \vee \neg q(y) \vee (p(z) \wedge q(z))$$

2. la transformation en forme de Skolem :

$$\forall y (\neg p(a) \vee \neg q(y) \vee (p(f(y)) \wedge q(f(y))))$$

où a est un symbole de fonction d'arité nulle (un symbole de constante) et f un symbole de fonction d'arité 1.

7.3.4 Démonstration

La démonstration informelle de l'équivalence de la consistance d'une formule et de sa forme de Skolem va nous permettre de donner également une interprétation intuitive des fonctionnelles apparaissant dans la forme de Skolem.

Nous allons nous restreindre aux formules sous forme prénexe ; soit F une telle formule et S_F sa forme de Skolem associée. Si F est consistante, il existe une interprétation I de F telle que $I(F) = t$, c'est-à-dire que pour tout choix de valeur des variables qui sont quantifiées universellement, je peux trouver une affectation des variables quantifiées existentiellement telles que la formule soit vraie. Dans la forme de Skolem, on remarque que chaque variable quantifiée existentiellement est remplacée par un symbole de fonction ne dépendant que des variables qui sont quantifiées universellement la précédant. Ce symbole de fonction doit donc s'interpréter comme étant la fonction de choix qui prend pour valeur précisément l'élément rendant la formule vraie.

Donc si F est satisfiable, S_F l'est aussi. La réciproque est immédiate : tout modèle de S_F est aussi un modèle de F.

7.3.5 Forme standard

La forme de Skolem d'une formule close ayant toujours toutes ses variables quantifiées universellement, les quantificateurs peuvent être omis dans l'énoncé de la forme. On parle alors *d'instance de la forme de Skolem.* Il ne reste alors plus qu'une étape pour amener la formule dans la forme qui nous intéressera dorénavant : passer à la forme clausale (ou standard).

La définition d'une clause (disjonction de littéraux) est exactement la même en calcul des prédicats et en calcul propositionnel. La méthode présentée au chapitre 6 s'applique directement à une instance de forme de Skolem (il n'y a plus de quantificateurs) et permet d'aboutir à la forme standard (équivalente à la forme de Skolem originale) :

Définition 7.3 – Forme standard – *Une forme standard est une matrice formée d'une conjonction de clauses.*

Nous manipulerons dorénavant toutes nos formules sous forme standard. Elles ne contiennent plus de quantificateurs, mais toute formule représente implicitement sa clôture universelle : toutes les variables sont liées.

7.4 Théorème de Herbrand

Nous allons tout d'abord définir la notion de terme de base et d'atome de base :

Définition 7.4 – Terme de base, atome de base – *On appelle terme de base un terme ne contenant pas de variable. De la même façon, un atome de base est un atome ne contenant pas de variable*[1].

Par exemple, $f(a)$ est un terme de base, alors que $f(x)$ n'en est pas un. De même, $p(a)$ est un atome de base, alors que $q(a, x)$ n'en est pas un.

Définition 7.5 – Univers de Herbrand – *On appelle univers de Herbrand d'un ensemble de clauses E, l'ensemble des termes de base que l'on peut construire à partir des symboles de fonctions et des symboles de constantes qui apparaissent dans E.*

Dans le cas où E ne contient aucun symbole de constante, on ajoute arbitrairement un symbole de constante a pour construire notre univers de Herbrand.

Considérons par exemple l'ensemble :

$$E = \{(p(f(x)) \rightarrow q(a)), r(g(x))\}$$

Ici, l'univers de Herbrand de E est $\{a, f(a), g(a), f(g(a)), g(f(a)), f(f(a)), \ldots\}$

Définition 7.6 – Base de Herbrand – *Soit un ensemble E de formules. La base de Herbrand de E est l'ensemble des atomes de bases qui peuvent être construits à partir des symboles de prédicats de E, appliqués aux termes de l'univers de Herbrand de E.*

Ainsi, dans l'exemple précédent, la base de Herbrand de E est

$$\{p(a), q(a), r(a), p(f(a)), q(f(a)), r(f(a)), p(g(a)), q(g(a)), \ldots\}$$

---

1 On parle aussi de terme (ou atome) *complètement instancié.*

Nous avons, dans le chapitre consacré au calcul des prédicats, montré ce qu'étaient une interprétation et un modèle d'une formule du calcul des prédicats. Nous allons maintenant introduire la notion d'interprétation et de modèle de Herbrand. Il est intéressant de la comparer à la définition 3.9 pour bien saisir la restriction faite par l'interprétation de Herbrand.

Définition 7.7 – Interprétation de Herbrand – *Dans une interprétation de Herbrand :*
- *l'ensemble de définition $\mathcal{D}$ (le domaine de l'interprétation) est l'univers de Herbrand de la formule F ;*
- *chaque symbole de constante de F est assigné à lui-même ;*
- *pour chaque symbole de fonction f d'arité n de F, le graphe de f est défini par : $(t_1, \ldots t_n) \rightarrow f(t_1, \ldots, t_n)$;*
- *pour un symbole de prédicat d'arité n de F, le graphe $\mathcal{D}^n \rightarrow \{t, f\}$ est quelconque.*

Une formule étant donnée, une interprétation de Herbrand peut se résumer par le sous-ensemble de la base de Herbrand dont les éléments sont interprétés à t.

Toute interprétation de Herbrand est bien une interprétation au sens où nous l'avons défini dans le chapitre 3. En revanche, une interprétation n'est pas toujours une interprétation de Herbrand.

Définition 7.8 – Modèle de Herbrand – *On appelle modèle de Herbrand une interprétation de Herbrand de F qui est un modèle de F.*

Théorème 7.2 – Théorème de Herbrand – *Un ensemble E de clauses est insatisfiable ssi il n'a pas de modèle de Herbrand.*

La preuve de ce théorème est relativement simple. D'une part, si E admet un modèle de Herbrand, alors il est consistant. D'autre part, si E admet un modèle M, il suffit de construire $M' = \{p(t_1, \ldots t_n) \in B_E \mid p(t_1, \ldots t_n)$ est vrai dans $M\}$. M' est clairement un modèle[2] de Herbrand de E.

Pour résumer, le théorème de Herbrand permet de réduire le nombre d'interprétations qu'il faut examiner pour démontrer la consistance ou l'inconsistance. La *base de Herbrand* est un domaine réduit dans lequel il suffit d'établir la consistance de la formule pour prouver sa consistance dans le cas général.

Pour donner une idée « intuitive » de ce qu'est une interprétation de Herbrand, on peut dire qu'il s'agit d'une interprétation propositionnelle de la formule. Il est d'ailleurs possible d'utiliser les algorithmes de la logique propositionnelle (algorithmes de Quine, de Davis et Putnam, ou l'algorithme de résolution) dans ce cadre. La possible non finitude de l'univers de Herbrand ne permet plus de garantir la terminaison de ces procédures. Pour les lecteurs intéressés, voir (Delahaye 1986; Thaysse and others 1988; Chang and Lee 1987; Lloyd 1987).

---

2 Attention ! Ce résultat n'est valable que parce que nous traitons un ensemble de *clauses*. Dans le cas général, le théorème de Herbrand ne s'applique pas. Considérons le cas trivial où $E = \{p(a), \exists x \neg p(x)\}$; il suffit de prendre comme domaine d'interprétation $\{a, b\}$ et de poser $p(a)$ vrai et $p(b)$ faux pour avoir un modèle de E. En revanche, les seules interprétations de Herbrand de E sont $\varnothing$ et $\{p(a)\}$ et aucune d'entre elles n'est un modèle de E.

7.5 Unification

7.5.1 Substitution

Il est temps de définir formellement la notion de substitution, que nous avons jusque-là utiliser un peu informellement. Substituer un terme t_1 à un terme t_2 dans une formule F consiste simplement à remplacer le terme t_2 par le terme t_1. Cependant, nous nous intéressons surtout à la substitution d'un terme aux variables (implicitement quantifiées universellement et donc liées) apparaissant dans les termes d'une forme clausale puisque cette opération conserve en partie l'interprétation sémantique de la dite formule. Ce type d'opération est appelé instanciation, et c'est elle que nous nous sommes permis d'effectuer jusqu'ici.

Définition 7.9 – Substitution – *Une substitution est une application de l'ensemble des variables dans l'ensemble des termes, presque partout égale à l'identité (elle n'en diffère qu'en un nombre fini de variables). On dira qu'une substitution σ est une substitution pure si pour toute variable x, $\sigma(x)$ est une variable.*

Une substitution σ est ainsi définie par l'ensemble *fini* des couples $(x, \sigma(x))$ pour tout x différent de $\sigma(x)$. On notera, abusivement, $\sigma = \cup\{x_i \leftarrow t_i\}$ pour chaque variable x_i telle que $\sigma(x_i) = t_i \neq x_i$.

On étend le domaine des substitutions aux termes via la définition d'une instance :

Définition 7.10 – Instance – *Le terme $\sigma(t)$ obtenu en remplaçant simultanément dans le terme t toutes les occurrences des variables x_i par $\sigma(x_i)$ est une instance de t.*

Un terme t_2 est une instance de t_1 ssi il existe une substitution σ tel que $t_2 = \sigma(t_1)$. On note alors $t_2 \prec t_1$.

Examinons ces définitions sur un exemple. Posons (ici x et y sont des variables et a une constante) :

- $\sigma(x) = a, \sigma(y) = f(x, a)$
- $t = g(x, y, f(x, y))$

Nous aurons alors :

$$\sigma(t) = g(a, f(x, a), f(a, f(x, a)))$$

Remarquons que $\sigma(t)$ est une instance de σ mais n'est pas un terme complètement instancié. Calculons $\sigma(\sigma(t))$:

$$\sigma(\sigma(t)) = g(a, f(a, a), f(a, f(a, a)))$$

Ce terme est lui, complètement instancié.

Remarquons également que la relation $\prec$ que nous avons définie est une relation de préordre (relation réflexive et transitive) sur l'ensemble des termes T.

Définition 7.11 – Instance fondamentale d'une clause – *Une instance d'une clause C dont tous les termes sont complètement instanciés est une instance de base, ou instance fondamentale, de C.*

Théorème 7.3 – Composition de substitutions – *Soit σ et θ, deux substitutions définies par $\sigma = \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$ et $\theta = \{y_1 \leftarrow u_1, \ldots, y_m \leftarrow u_m\}$, la composition de σ et de θ est la substitution $\theta \circ \sigma$ définie à partir de l'ensemble*

$$\{x_1 \leftarrow \theta(t_1), \ldots, x_n \leftarrow \theta(t_n), y_1 \leftarrow u_1, \ldots, y_m \leftarrow u_m\}$$

dont on a supprimé tous les $x_i \leftarrow \theta(t_i)$ *pour lesquels* $x_i = \theta(t_i)$ *et tous les* $y_i \leftarrow u_i$ *tels que* $y_i \in \{x_1, \ldots, x_n\}$.

Exemple : considérons σ définie par $\{x \leftarrow y, y \leftarrow f(y)\}$ et la substitution pure θ définie par $\{y \leftarrow x\}$. La substitution $\theta \circ \sigma$ est alors définie par l'ensemble $\{y \leftarrow f(x)\}$ (on a supprimé $y \leftarrow x$ car y est une variable sur laquelle σ n'est pas l'identité et $x \leftarrow \theta(y)$ car $\theta(y)$ est égale à x).

Nous définissons en passant les substitutions de renommage, permettant de définir précisément ce qu'est un renommage de variable.

Définition 7.12 – Substitution de renommage – *Soit A une formule et V l'ensemble des variables apparaissant dans A. Une substitution de renommage de A est une substitution pure* $\sigma = \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$ *telle que* $\{x_1, \ldots, x_n\} \subset V$, *tous les* t_i *sont distincts et* $(V - \{x_1, \ldots, x_n\}) \cap \{t_1, \ldots, t_n\} = \varnothing$.

7.5.2 Unification

Définition 7.13 – Littéraux unifiables – *Deux termes sont dits unifiables s'ils possèdent une instance fondamentale commune.*

Définition 7.14 – Unificateur – *Soit un ensemble* $E = \{t_1, \ldots, t_n\}$ *de termes. On appelle unificateur de E toute substitution* σ *telle que*

$$\sigma(s_1) = \sigma(s_2) = \cdots = \sigma(s_n)$$

Définition 7.15 – Unificateur le plus général – *On appelle unificateur le plus général d'un ensemble E d'expressions, un unificateur* θ *de E (s'il existe) tel que pour tout autre unificateur* σ *de E il existe une substitution* σ' *telle que* $\sigma = \sigma' \circ \theta$.

Notons qu'il n'existe pas forcément *un seul* plus général unificateur d'un ensemble E. Considérons par exemple $E = \{y, f(x)\}$. Les substitutions θ et θ' définies respectivement par $\{y \leftarrow f(x)\}$ et par $\{x \leftarrow y, y \leftarrow f(y)\}$ sont deux plus généraux unificateurs de E.

En revanche, on peut montrer que tous les plus généraux unificateurs sont équivalents modulo l'application de substitutions pures[3]. Dans l'exemple précédent, si l'on considère la substitution pure σ définie par $\{y \leftarrow x\}$, on a effectivement $\theta = \sigma \circ \theta'$ comme on a pu le voir précédemment.

---

3 La procédure la plus classique pour calculer un des plus généraux unificateurs d'un ensemble de termes, classiquement désignée sous le nom d'algorithme de Robinson (Robinson 1965), calcule en fait un plus général unificateur θ idempotent ($\theta \circ \theta = \theta$) et a une complexité exponentielle dans le pire des cas. L'unification est en fait un problème linéaire comme l'ont montré Patterson et Wegman (Paterson and Wegman 1978). Depuis, d'autres algorithmes plus efficaces en moyenne ont été développés (Escalada and Ghallab 1987).

7.6 Principe de résolution

Nous présentons ici un mécanisme de résolution qui dérive directement de la méthode introduite en calcul propositionnel, mais utilise l'opération d'unification pour ne pas opérer sur les instances fondamentales des clauses, mais sur les clauses elles-mêmes.

Établissons tout d'abord le principe de résolution (c'est une règle d'inférence quelque peu complexe) :

Théorème 7.4 – Principe de résolution – *Soit N une forme normale, C_1 et C_2 deux clauses de N. Soit deux littéraux $l_1 \in C_1$ et $\neg l_2 \in C_2$, θ une substitution de renommage telle que $\theta(C_1)$ et C_2 n'aient aucune variable libre commune. Soit σ, un plus grand unificateur de $\theta(C_1)$ et C_2, et R la clause*

$$R = \sigma(\theta(C_1 \setminus \{l_1\}) \vee (C_2 \setminus \{\neg l_2\})$$

appelée résolvante. Alors N et $N \cup R$ sont logiquement équivalentes.

Ce théorème se déduit du théorème de résolution pour le calcul propositionnel. Il définit une première règle d'inférence. Ce principe ne suffit pas, seul, à assurer que l'on pourra inférer la clause vide quand elle est conséquence logique d'un ensemble de clauses. On lui adjoint une règle de factorisation :

Théorème 7.5 – Règle de factorisation – *Soit C_1 une clause de la forme $l_1 \vee l_2 \vee \ldots l_n$ telle que l_1 et l_2 sont deux littéraux unifiables. Soit σ, un plus grand unificateur de l_1 et l_2. La clause $C'_1 = \sigma(C_1 \setminus \{l_2\})$ est une conséquence logique de C_1. C'_1 est appelée facteur de C_1.*

Ce théorème définit une règle d'inférence qui est indispensable à la bonne utilisation du principe de résolution sur des formes clausales. Sans cette règle, il serait impossible d'inférer la clause vide à partir des deux clauses $p(x) \vee p(a)$ et $\neg p(x) \vee \neg p(a)$ car le principe de résolution construit une résolvante contenant deux littéraux de moins que la somme du nombre de littéraux des clauses résolues, ici *toujours* 2.

Le système formé d'un ensemble d'axiomes vide et de ces deux règles d'inférence est alors complet pour la réfutation *i.e.*, si la clause vide est conséquence d'un ensemble de clauses (qui sera alors inconsistant), alors elle sera théorème de notre système de déduction.

7.6.1 Algorithme et exemple d'application

L'algorithme de résolution est très simple, et directement calqué sur l'algorithme de résolution pour le calcul propositionnel.

Soit N une forme normale :

1. si $\varnothing \in N$, alors la résolution est terminée ;
2. prendre C_1 et C_2 deux clauses ou facteurs de clauses dans N, telles que $l_1 \in C_1$, $\neg l_2 \in C_2$ et l_1, l_2 sont unifiables ;
3. trouver un plus grand unificateur σ de l_1 et l_2, et calculer la clause résolvante R[4] ;

---

4 Signalons tout de même un « petit problème » bien connu de tous les gens qui s'intéressent à la résolution automatique. Supposons que nous ayons à unifier des termes x et $f(x)$. L'algorithme d'unification pourrait retourner

4. remplacer N par $N \cup \{R\}$.

Nous allons prendre un exemple classique consistant à montrer que si dans un groupe, chaque élément est son propre inverse, alors le groupe est commutatif. L'exemple est pris dans (Thaysse and others 1988).

Nous avons trois hypothèses :

1. $\forall x \forall y \forall z, (xy)z = x(yz)$: associativité ;
2. $\forall x, xe = ex = x$: élément neutre e ;
3. $\forall x, xx = e$: chaque élément est son propre inverse.

Nous voulons arriver à la conclusion C :

$$\forall x, \forall y, xy = yx$$

On introduit un prédicat $E(x, y, z)$ à trois variables qui s'interprète comme $(xy = z)$[5].

Il ne nous reste plus qu'à mettre notre système sous forme normale (skolémisation, etc.). Nous obtenons alors :

1. $\neg E(x, y, u) \vee \neg E(y, z, v) \vee \neg E(u, z, w) \vee E(x, v, w)$
2. $\neg E(x, y, u) \vee \neg E(y, z, v) \vee \neg E(x, v, w) \vee E(u, z, w)$
3. $E(x, e, x)$
4. $E(e, x, x)$
5. $E(x, x, e)$
6. $E(a, b, c)$
7. $\neg E(b, a, c)$

(3) exprime $x = xe$, (4) $x = ex$, (5) $e = xx$, et (6) et (7) sont la négation de la conclusion $(c = ab) \wedge (c \neq ba)$. Il est intéressant de détailler (1) et (2) pour montrer (informellement) qu'elles expriment l'associativité. Remarquons que l'associativité peut aussi s'exprimer par :

$$(u = xy) \wedge (v = yz) \wedge (w = uz) \wedge (w = xv)$$

Nous pouvons alors constater que l'on peut réécrire cette formule en :

$$\neg(((u = xy) \wedge (v = yz) \wedge (w = uz) \wedge (w \neq xv)) \vee ((u = xy) \wedge (v = yz) \wedge (w = xv) \wedge (w \neq uz)))$$

---

l'unifieur défini par $\sigma(x) = f(f(f(f(\ldots(x)\ldots))))$. Mais l'univers de Herbrand n'est pas censé contenir de termes infinis (on peut éventuellement décider, comme le fait PROLOG II de travailler sur un domaine plus riche, contenant des termes dits rationnels et correspondant à des « arbres infinis »). En toute rigueur, il faudrait donc vérifier avant chaque unification que l'on ne risque pas de se trouver dans une situation de ce type *i.e.*, que l'on n'essaie pas d'unifier une variable x avec un terme contenant x, mais ne se limitant pas à x. Ce test, qu'il faudrait effectuer, s'appelle *test d'occurrence*, ou *occur-check* en anglais. Aucun des langages de programmation logique actuels n'effectue le test d'occurrence.

5 Nous aurions pu dans notre axiomatique du calcul des prédicats, présenter l'axiomatique du calcul des prédicats avec l'égalité. Voir (Kleene 1987).

En passant la négation dans la parenthèse :

$$((u \neq xy) \vee (v \neq yz) \vee (w \neq uz) \vee (w = xv)) \wedge$$
$$((u \neq xy) \vee (v \neq yz) \vee (w \neq xv) \vee (w = uz))$$

Ce qui nous ramène bien à deux clauses :
- $\neg E(x, y, u) \vee \neg E(y, z, v) \vee \neg E(u, z, w) \vee E(x, v, w)$
- $\neg E(x, y, u) \vee \neg E(y, z, v) \vee \neg E(x, v, w) \vee E(u, z, w)$

Nous présentons ci-dessous la résolution :

8. $\neg E(x, z, v) \vee \neg E(e, z, w) \vee E(x, v, w)$: en unifiant le premier littéral de (1) et (5). On renomme x en x' dans (1). L'unification produit la substitution $\sigma = \{y \leftarrow x, u \leftarrow e, x' \leftarrow x)\}$.
9. $\neg E(b, z, v) \vee \neg E(a, v, w) \vee E(c, z, w)$: premier littéral de (2) et (6). $\sigma = \{x \leftarrow a, y \leftarrow b, u \leftarrow c)\}$.
10. $\neg E(x', z, v) \vee E(x', v, z)$: deuxième littéral de (8) et (4). Ici encore, la variable x apparaît libre dans les deux clauses. On renomme donc x en x' par une substitution de renommage appliquée à (8). Le plus général unificateur utilisé est $\{z \leftarrow x, w \leftarrow x\}$.
11. $\neg E(a, e, w) \vee E(c, b, w)$: premier littéral de (9) et (5). Unification via $\{x \leftarrow b, z \leftarrow b, v \leftarrow e\}$.
12. $E(c, b, a)$: (3) et (11)
13. $E(c, a, b)$: (10) et (12)
14. $\neg E(x, y, u) \vee \neg E(x, e, w) \vee E(u, y, w)$: (5) et le deuxième littéral de (2).
15. $\neg E(x, y, u) \vee E(u, y, x)$: deuxième littéral de (14) et (3)
16. $E(b, a, c)$: (15) et (13)
17. $\varnothing$: (16) et (7)

Q.E.D

CHAPITRE 8

Les logiques non-classiques

Jean-Marc Alliot

8.1 Logiques faibles

Avant d'aborder l'étude des systèmes modaux, nous allons dire un mot des systèmes logiques proches de la logique classique, qui dérivent d'une interprétation plus faible de la notion de négation ou d'implication. Nous reprenons ici la présentation faite dans (Grize 1967).

Plutôt que de choisir un nombre minimal de schémas d'axiomes n'utilisant que les symboles $\rightarrow$ et $\neg$ puis de définir tous les connecteurs à partir de ces deux-là comme nous l'avions fait lors de la présentation de la logique classique propositionnelle, il est possible de partir d'un système définissant chacun des connecteurs logiques séparément. Ceci est même nécessaire, dans la mesure où ces connecteurs ne sont plus inter-définissables dans les logiques considérées dans cette section.

8.1.1 La logique absolue A

Les schémas d'axiomes de A sont :

- pour $\rightarrow$:

Axiome 8.1 – A1 –

$$P \rightarrow (Q \rightarrow P)$$

Axiome 8.2 – A2 –

$$(P \rightarrow (Q \rightarrow M)) \rightarrow ((P \rightarrow Q) \rightarrow (P \rightarrow M))$$

On reconnaît les axiomes 2.1 et 2.2 de la logique classique.

- pour $\wedge$:

Axiome 8.3 – A3 –

$$(P \wedge Q) \rightarrow P$$

Axiome 8.4 – A4 –

$$(P \wedge Q) \rightarrow Q$$

Axiome 8.5 – A5 –

$$(P \rightarrow Q) \rightarrow ((P \rightarrow M) \rightarrow (P \rightarrow (Q \wedge M)))$$

– pour $\vee$:

Axiome 8.6 – A6 –

$$P \rightarrow (P \vee Q)$$

Axiome 8.7 – A7 –

$$Q \rightarrow (P \vee Q)$$

Axiome 8.8 – A8 –

$$(P \rightarrow M) \rightarrow ((Q \rightarrow M) \rightarrow ((P \vee Q) \rightarrow M))$$

Le signe $\neg$ n'apparaît nulle part dans les schémas d'axiomes de A. La négation n'existe pas dans ce système.

8.1.2 La logique positive P

Le système A est un système extrêmement faible. Non seulement la négation n'existe pas dans A, mais certains théorèmes de la logique classique, comme $p \vee (p \rightarrow q)$ qui ne contient pourtant pas de signe de négation, ne sont pas des théorèmes de A. Si l'on rajoute l'axiome[1] :

Axiome 8.9 – A9 –

$$A \vee (A \rightarrow B)$$

le système ainsi défini contient bien l'ensemble des théorèmes de la logique classique ne contenant aucune négation. On l'appelle logique positive.

---

1 Nous confondrons dans la suite axiome et schéma d'axiome.

8.1.3 La logique minimale M

Si l'on étend le système A par les deux axiomes suivants :

Axiome 8.10 – A10 –

$$\neg A \rightarrow (A \rightarrow \neg B)$$

Axiome 8.11 – A11 –

$$(A \rightarrow \neg A) \rightarrow \neg A$$

on obtient la logique minimale M. Dans ce système, le symbole de négation a pour interprétation « est réfutable »[2]. Dans M, on a le théorème :

Théorème 8.1 – Négation et implication –

$$p \rightarrow \neg\neg p$$

En revanche, la réciproque est fausse.

8.1.4 La logique intuitionniste J

Si l'on considère le système M, qu'on lui retranche l'axiome A10 et qu'on le remplace par :

Axiome 8.12 – A12 –

$$\neg A \rightarrow (A \rightarrow B)$$

on obtient la logique intuitionniste. Tout théorème de M est évidemment un théorème de J (A10 est une conséquence triviale de A12). Dans cette logique, le symbole de négation peut se comprendre comme « est absurde ».

La logique classique se retrouve en adjoignant à la logique intuitionniste le schéma d'axiome A9.

8.2 Introduction aux logiques modales

Les problèmes concernant la notion de devoir et de nécessité sont apparus très tôt. Husserl résumera le problème :

> « Les énoncés négatifs du devoir ne doivent pas être interprétés comme des négations des énoncés positifs correspondants. »

---

2 Il serait trop long de développer complètement la notion de réfutabilité. Disons simplement que dans un tel système, il existe à côté des théorèmes (vrais) des anti-théorèmes (faux). Une proposition p est réfutable ($\neg p$ est vrai) si l'on peut démontrer que p conduit, à l'aide des axiomes, à l'un de ces anti-théorèmes.

Le logicien anglais Lewis s'intéressera aux notions de nécessité et de possibilité à travers un des problèmes intuitifs lié à la logique classique, la formule :

$$\neg A \rightarrow (A \rightarrow B)$$

Cette formule, qui se déduit aisément des axiomes de la logique classique, se lit : « si A est faux, je peux déduire n'importe quoi de A ».

D'autres formules, comme $A \rightarrow (B \rightarrow A)$[3], ont posé très rapidement des problèmes aux logiciens. Pour tenter de résoudre ces « paradoxes », Lewis proposa au début du siècle une nouvelle implication, dite implication stricte, qu'il nota : $>$.

Pour cela, il définit la notion de possibilité logique qu'il nota $\Diamond$ et posa par définition :

$$(A > B) =_{def} \neg\Diamond(A \wedge \neg B)$$

Ceci peut se lire : « il est impossible que A soit vrai et B faux lorsque A implique strictement B ».

Il définit également la notion (duale) de nécessité, notée $\Box$ par :

$$\Box A =_{def} \neg\Diamond\neg A$$

On peut le lire : « A est nécessaire si non-A est impossible ».

Muni de ces définitions et de ces deux nouveaux opérateurs, dits opérateurs modaux, Lewis définit deux systèmes, S4 et S5, que nous allons examiner brièvement. Dans chacun de ces systèmes, l'implication stricte définie précédemment n'est plus paradoxale. En effet, les formules $\neg A > (A > B)$ et $A > (B > A)$ ne sont pas toujours des théorèmes de ces systèmes.

8.3 Les modalités

Les notions de possibilité et de nécessité sont des instances particulières de la notion plus générale de *modalité.*

Définition 8.1 – Modalité – *On appelle modalité toute suite d'opérateurs* $\neg$, $\Box$, $\Diamond$.

Par exemple $\neg\Box\Box\neg\Diamond$ est une modalité.

On voit bien intuitivement que des modalités syntaxiquement distinctes peuvent avoir la même signification. Ainsi les modalités $\neg\Diamond$ et $\Box\neg$ sont équivalentes dans tous les systèmes présentés ici.

Définition 8.2 – Modalités équivalentes – *Deux modalités* α *et* β *sont équivalentes si, pour toute formule A, on a :* $\vdash (\alpha A \leftrightarrow \beta A)$

Une logique modale peut être caractérisée par l'ensemble de ses modalités non-équivalentes. Pour établir la liste des modalités non-équivalentes, on tente d'établir l'ensemble des théorèmes de réduction syntaxique démontrables dans la logique dont on se préoccupe. Nous démontrerons un tel théorème dans S4.

---

3 que l'on peut lire : « Si A est vrai alors A se déduit de n'importe quoi ».

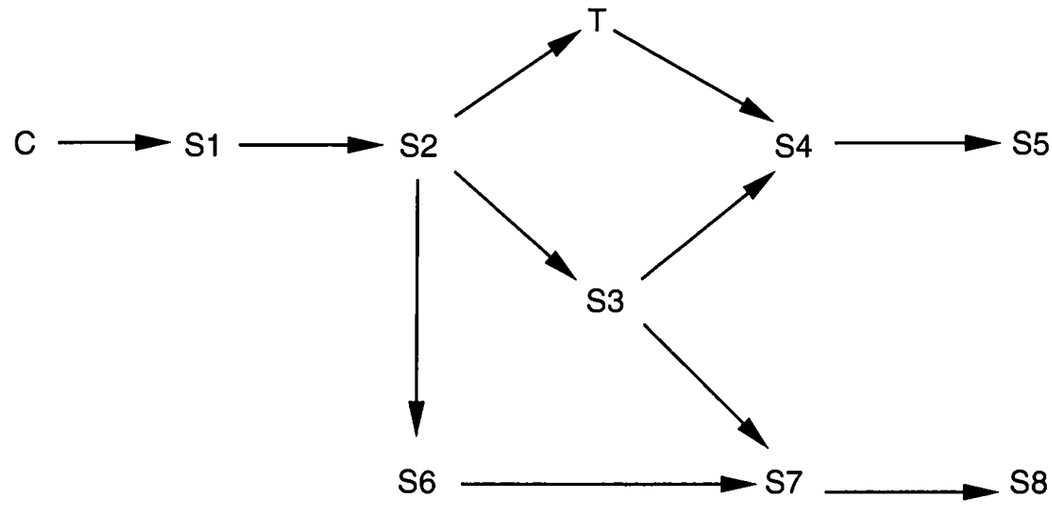


Figure 8.1 – Ensemble des systèmes en logique aléthique

8.4 Logique aléthique

On appelle logique aléthique toute logique dont les modes sont la *possibilité* et la *nécessité*. Les logiques définies par Lewis sont des logiques aléthiques. Le langage des premiers systèmes que nous allons étudier est construit comme celui du calcul propositionnel en y adjoignant la règle :

Définition 8.3 – *Si A est une formule, alors $\Diamond A$ et $\Box A$ sont des formules.*

La construction faite par Lewis est un peu différente de celle que nous allons donner. Les systèmes de logique aléthique sont représentés dans la figure 8.1. Dans cette figure (due à Prior), tous les systèmes qui sont à gauche des flèches sont des sous-systèmes des systèmes situés à droite, c'est-à-dire que tout théorème valable dans le système de gauche est aussi un théorème du système de droite.

Les systèmes S1 à S5 sont les systèmes définis par Lewis. Le système T est dû à Freys (1950) et Wright (1951), et est donc bien plus tardif. Cependant, il est formellement plus agréable d'arriver à S4 et S5 à partir de T qu'à partir de S1 et S2.

8.4.1 Le système T

Le système T est défini par les axiomes de la logique classique et les axiomes supplémentaires suivants :

Axiome 8.13 – K –

$$\Box(A \to B) \to (\Box A \to \Box B)$$

Axiome 8.14 – T –

$$\Box A \to A$$

On adjoint à la règle d'inférence de la logique classique la règle :

Règle d'inférence 8.1 – Règle de nécessitation –

$$\frac{\vdash A}{\vdash \Box A}$$

T contient une infinité de modalités non-équivalentes. Il est consistant et il existe une procédure de décision.

8.4.2 Le système S4

Les règles d'inférence de S4 sont les mêmes que celles de T. On ajoute simplement l'axiome :

Axiome 8.15 – 4 – $\Box A \rightarrow \Box\Box A$

Nous allons démontrer rapidement un théorème de réduction dans S4.

Théorème 8.2 – Réduction de $\Box\Box$ en $\Box$ –

$$\vdash \Box\Box A \leftrightarrow \Box A$$

1. $\vdash \Box\Box A \rightarrow \Box A$ Axiome T et substitution de $\Box A$ à A
2. $\vdash \Box A \rightarrow \Box\Box A$ Axiome 4
3. $\vdash \Box\Box A \leftrightarrow \Box A$

On peut ainsi démontrer que dans S4 toute modalité ne contenant pas $\neg$ peut être réduite à une suite contenant au plus 3 opérateurs modaux. Notons enfin que S4 est complet et décidable.

8.4.3 Le système S5

S5 est également une extension de T. Il suffit de rajouter l'axiome suivant :

Axiome 8.16 – 5 –

$$\Diamond A \rightarrow \Box\Diamond A$$

D'autre part, tout théorème de S4 est un théorème de S5. En effet, il est aisé de montrer que l'axiome 8.15 est un théorème de S5. Enfin, on peut démontrer que, dans S5, toute modalité ne contenant pas de négation peut être réduite à un opérateur (théorème de réduction).

8.5 Théorie des modèles

Nous nous sommes intéressés jusque-là à la théorie de la démonstration pour les logiques non-classiques. La question que nous sommes en droit de nous poser est la suivante : est-il possible de construire une théorie des modèles (semblable à la méthode des tables de vérité) dans un système comme S4 ou S5 ?

8.5.1 Les tables de vérité

Nous devons tout d'abord constater que la méthode des tables de vérité qui consiste à associer à un opérateur une table est ici inapplicable. En effet, considérons qu'il fait beau aujourd'hui ; alors l'énoncé : « Le ciel est bleu » est vrai, ainsi que l'énoncé : « Le ciel est bleu ou le ciel n'est pas bleu ». En revanche, l'énoncé : « Le ciel est nécessairement bleu » est faux, car il peut ne pas faire beau, alors que l'énoncé : « Nécessairement, le ciel est bleu ou le ciel n'est pas bleu » est vrai. On voit donc que l'énoncé $\Box A$ peut être vrai ou faux, alors que l'énoncé A est vrai dans les deux cas. On voit donc mal comment la méthode des tables de vérité pourrait être appliquée.

8.5.2 Sémantique des mondes possibles

Définition 8.4 – Modèle de Kripke – *Un modèle de Kripke est défini par un triplet* $M =< W, \mathcal{R}, m >$ *où* $W = \{w_1, w_2, \ldots, w_i \ldots\}$ *est un ensemble de mondes possibles*[4], $\mathcal{R}$ *est une relation binaire définie sur* W *et* m *une fonction associant à chaque élément de* W *un sous-ensemble de* V_p *(ensemble des variables propositionnelles).*

Définition 8.5 – Relation de satisfiabilité – *Si* A *est une formule,* M *un modèle, et* w *un monde possible de* M*, alors la relation « Le monde* w *satisfait* A *», notée* $(M, w \models A)$*, est définie par :*

- $M, w \models p$ *ssi* $p \in m(w)$
- $M, w \models \neg A$ *ssi non*$(M, w \models A)$
- $M, w \models (A \vee B)$ *ssi* $(M, w \models A)$ *ou* $(M, w \models B)$
- $M, w \models (A \wedge B)$ *ssi* $(M, w \models A)$ *et* $(M, w \models B)$
- $M, w \models \Box A$ *ssi quel que soit* w_1 *tel que* $w \mathcal{R} w_1$ *on a* $M, w_1 \models A$
- $M, w \models \Diamond A$ *ssi il existe* w_1 *tel que* $w \mathcal{R} w_1$ *et* $M, w_1 \models A$

Définition 8.6 – Formule satisfiable – *Une formule* A *est dite satisfiable ssi il existe un modèle* M *et un état* w *de* M *tel que* $M, w \models A$.

Définition 8.7 – Formule valide – *Une formule* A *est dite valide (et l'on note* $\models A$*) ssi pour tout* M *et pour tout* w *de* M *on a* $M, w \models A$.

Ainsi, en considérant une interprétation comme un couple (M, w), une formule valide est une formule vraie pour toute interprétation, ce qui est bien compatible avec la définition que nous avions utilisée en logique propositionnelle et en calcul des prédicats.

8.5.3 Interprétation intuitive

Que cherchons-nous à exprimer par les définitions précédentes ? Nous avons vu que la vérité d'une proposition du type $\Box A$ semblait être liée à une notion d'universalité de la vérité de l'énoncé A. Nous allons donc définir un ensemble de mondes contenant chacun des formules, puis nous les relierons entre eux par une relation d'*accessibilité* de la connaissance ; si par exemple $w_1 \mathcal{R} w_2$ et $w_1 \mathcal{R} w_1$ alors pour que $M, w_1 \models \Box p$ il faudra que $p \in m(w_1)$ et $p \in m(w_2)$. Pour avoir $M, w_1 \models p$, il suffit que $p \in m(w_1)$ et pour que $M, w_1 \models \Diamond p$ il suffit que $p \in m(w_1)$ ou $p \in m(w_2)$.

---

4 W peut être fini ou infini.

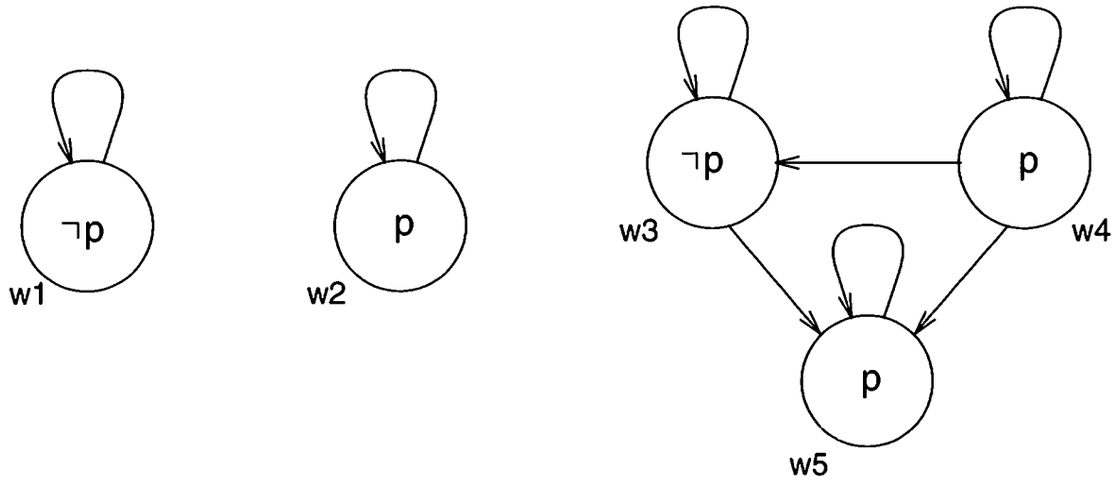


Figure 8.2 – Axiome T et réflexivité

On voit donc qu'il est possible par ce moyen de construire un modèle pour tester la validité d'une formule. Cependant, ce modèle n'est pas quelconque. En fait les axiomes de la logique déterminent le type de la relation d'accessibilité $\mathcal{R}$.

8.5.4 Un modèle de S4

Définition 8.8 – Modèle de S4 – *Un modèle de Kripke pour S4 est un modèle* $M =< W, \mathcal{R}, m >$ *où* $\mathcal{R}$ *est une relation réflexive et transitive.*

Nous allons essayer de donner une idée intuitive de la démonstration pour la propriété de réflexivité.

En fait, la réflexivité est rendue nécessaire par l'axiome T ($\Box A \rightarrow A$). Examinons la figure 8.2. Nous devons aisément comprendre à partir de cette figure pourquoi l'axiome T est vrai dans une structure réflexive.

- remarquons tout d'abord que si pour un monde w, $p \in m(w)$, alors $\Box p \rightarrow p$ est vérifié que $\Box p$ le soit ou non (logique classique) ;
- en revanche, si $p \notin m(w)$, alors, comme la relation est réflexive, $\Box p$ n'est pas vérifié et la formule $\Box p \rightarrow p$ est vérifiée.

8.5.5 Un modèle de S5

On peut démontrer que la relation d'accessibilité de S5 doit être une relation d'équivalence.

8.6 Adéquation, consistance, complétude

Comme en logique classique, on peut s'intéresser en logique modale aux propriétés de consistance et de complétude.

Le but de cet ouvrage n'étant pas de développer une théorie générale des logiques modales, nous nous contenterons de citer les principaux résultats. Les lecteurs intéressés pourront se reporter à (Hughes and Cresswell 1972).

Théorème 8.3 – *S4 et S5 sont syntaxiquement consistants, adéquats et complets vis-à-vis de l'interprétation sémantique définie par les modèles de Kripke.*

Il faut noter que, pour chacun de ces systèmes, on peut rajouter certaines formules qui ne sont pas des théorèmes de ces systèmes en tant que schémas d'axiomes sans aboutir à des systèmes syntaxiquement inconsistants[5]. En effet, nous avons vu que T peut-être étendu à S4, que S4 peut être étendu à S5. D'autre part, il est toujours possible de rajouter à S5 le schéma :

$$A \rightarrow \Box A$$

Ce schéma ne rend pas le système inconsistant, et n'est pas non plus un théorème de S5[6].

8.7 Une logique modale du premier ordre

Comme nous l'avons fait pour le calcul propositionnel, il est possible de définir une logique modale opérant sur des prédicats. La définition du langage est la même que pour la logique des prédicats avec l'adjonction de la définition suivante :

Définition 8.9 – *Si A est une formule alors $\Box A$ et $\Diamond A$ sont des formules.*

On peut ainsi définir des systèmes S4 et S5 du premier ordre en adjoignant à l'axiomatique propositionnelle de S4 et S5, les axiomes de la logique des prédicats (voir (Hughes and Cresswell 1972)).

8.8 Interprétation des logiques modales

Les logiques épistémiques sont des logiques ayant pour but de représenter la notion de connaissance ou de croyance. Elles ont été introduites par J. Hintikka en 1963.

8.8.1 Une logique de la connaissance

La logique de la connaissance est basée sur S5. Les opérateurs modaux rencontrés prennent alors une autre signification ; ainsi $\Box$ devient « SAVOIR » et $\Diamond$ devient « COMPATIBLE AVEC MES CONNAISSANCES ».

Examinons les axiomes et les règles d'inférence de S5 à la lumière de ces nouvelles définitions.

La règle d'inférence :

$$\frac{\vdash A}{\vdash \Box A}$$

peut alors se lire : « si A est démontrable, (je sais A) est démontrable[7] ».

---

5 On dit d'un tel système qu'il n'est pas *complet* au sens de Post.

6 Il est intéressant de noter que dans ce système, on a de façon triviale $A \leftrightarrow \Box A$ et $A \leftrightarrow \Diamond A$. Ce système est en fait semblable au calcul propositionnel puisque les opérateurs modaux peuvent être insérés ou supprimés à volonté, et n'ont donc aucune signification particulière.

7 On désigne parfois cette propriété sous le nom d'*omniscience*.

L'axiome $\Box(A \rightarrow B) \rightarrow (\Box A \rightarrow \Box B)$ peut se lire : « si je sais que $A \rightarrow B$ est vrai, alors, si je sais que A est vrai, je sais que B est vrai ».

L'axiome $\Box A \rightarrow A$ devient : « si je sais que A est vrai, alors A est vrai ».

L'axiome $\Diamond A \rightarrow \Box\Diamond A$ devient : « si A est compatible avec mes connaissances, alors je sais que A est compatible avec mes connaissances ». Cet axiome est un axiome dit *d'introspection négative.*

On remarquera que la propriété de S5 : $\Box A \rightarrow \Box\Box A$, est également une propriété d'introspection, mais positive[8].

Cette logique est appelée logique de la connaissance en raison de l'axiome $\Box A \rightarrow A$, qui exprime que si l'on connaît une propriété alors cette propriété est vraie.

8.8.2 Une logique de la croyance

La logique de la croyance est obtenue à partir des axiomes de S5, en supprimant l'axiome $\Box A \rightarrow A$ et en rajoutant dans le jeu des axiomes la propriété : $\Box A \rightarrow \Box\Box A$.

La suppression de $\Box A \rightarrow A$ reflète le fait que le concept de croyance admet que l'on peut croire des choses qui sont fausses dans la réalité.

8.9 Extensions

Une extension classique valable pour toutes les logiques épistémiques est l'introduction d'arguments dans les opérateurs modaux. Ainsi, on peut écrire $\Box[\text{Pierre}]A$, qui se lira « Pierre sait que A est vrai ». Ces systèmes permettent de représenter des environnements où les connaissances sont disjointes. On peut par exemple écrire : $\Box[\text{Pierre}]A \rightarrow \Box[\text{Paul}]A$, que l'on peut traduire par : « Paul sait tout ce que sait Pierre ». De telles logiques sont appelées multi-modales.

L'extension au premier ordre permet, elle, d'exprimer des propositions de la forme :

$$\exists x \Box \text{espion}(x)$$

qui signifie : « il existe une personne x dont je sais que c'est un espion » à ne pas confondre avec cette autre proposition également exprimable :

$$\Box\exists x \ \text{espion}(x)$$

qui signifie, elle : « je sais qu'il existe un espion (mais je ne sais pas qui c'est) ».

8.10 Les logiques temporelles

Nous nous contenterons dans ce chapitre de donner un aperçu des logiques temporelles. Les lecteurs intéressés peuvent se reporter pour plus de précisions à (Audureau *et al.* 1990), dont nous nous sommes largement inspirés.

---

8 La propriété d'introspection négative $\Diamond A \rightarrow \Box\Diamond A$ est équivalente à $\neg\Box A \rightarrow \Box\neg\Box A$. Ceci permet de mieux comprendre le nom d'introspection négative, par opposition à l'introspection positive.

8.10.1 Approche intuitive

Le but de la logique temporelle est d'enrichir (comme le fait la logique modale) la capacité d'interprétation de la logique classique.

Ainsi l'énoncé « Robespierre est mort », s'il est vrai aujourd'hui, était faux en 1789. Or, ce type d'évaluation est impossible en logique classique. Là où la logique modale a introduit des notions de nécessité et de possibilité, la logique temporelle va introduire les notions de futur et de passé.

Il existe deux façons de définir une logique temporelle. L'une d'entre elles est basée sur les dates. On introduit un opérateur binaire de datation, généralement noté $\mathcal{R}$. L'expression $\mathcal{R}tp$ signifiant « p est réalisé à l'instant t ». De cette façon , on peut aisément en fixant une échelle et une origine des temps, exprimer des propriétés comme « Hitler est mort en 1945 ». En revanche, cette méthode ne nous permet pas de représenter simplement des énoncés comme « le temps sera beau ». Pour représenter ce type d'énoncé, nous avons besoin d'un opérateur dit *opérateur d'ultériorité*, noté généralement $\mathcal{U}$. Ainsi on écrira : $\mathcal{R}t(\mathcal{U}tt' \wedge pt')$ avec $\mathcal{U}tt'$ signifiant « t' est ultérieur à t » et p étant « le temps est beau ». Il faut remarquer que cette seconde méthode repose sur ce que l'on appelle des *pseudo-dates*. En effet, nous utilisons comme éléments de référence des termes comme « aujourd'hui » ou « demain » qui ne sont pas à proprement parler des dates (c'est-à-dire des éléments fixes). Les premiers travaux de la logique temporelle furent en grande partie consacrés à établir que le calcul fondé sur les pseudo-dates pouvait se dériver du calcul sur les dates.

8.10.2 Premiers éléments

Les premiers opérateurs que nous allons définir seront les opérateurs F et P, opérateurs *futur* et *passé*. La définition de ces opérateurs en langage des pseudo-dates est[9] :

1. $Fp = \exists t(\mathcal{U}t't \wedge pt)$
2. $Pp = \exists t(\mathcal{U}tt' \wedge pt)$

On peut donner comme interprétation de Fp : « p sera vrai » et comme interprétation de Pp : « p a été vrai ».

On peut, à l'aide de ces opérateurs, exprimer les notions d'ultériorité. Ainsi si on veut exprimer que p est postérieur à q, on peut écrire :

$$(p \wedge (\neg q \wedge Pq)) \vee P(p \wedge (\neg q \wedge Pq)) \vee F(p \wedge (\neg q \wedge Pq))$$

On peut remarquer les trois parties de cette formule. La première $(p \wedge (\neg q \wedge Pq))$ signifie que p est vrai, que q est faux, et que q a été vrai dans le passé. Les deux autres parties complètent la première en exprimant que ce « fait » a été vrai dans le passé ou qu'il sera vrai dans le futur. Ainsi, la composition des trois permet de dire que p est postérieur à q aujourd'hui ou qu'il le sera ou qu'il l'a été (informellement).

On voit immédiatement que, comme en logique modale, la méthode des tables de vérité n'est pas applicable en logique temporelle.

---

9 Ici t' est interprété comme « aujourd'hui ».

En fait, il existe une forte similitude entre logique modale et logique temporelle. En théorie des modèles de la logique modale, nous avons vu qu'un modèle était un ensemble de mondes, et que la relation d'accessibilité des connaissances entre ces mondes déterminait le type de la logique.

En logique temporelle, les mondes possibles deviennent les mondes datés et la relation d'accessibilité devient la relation d'ultériorité entre les dates. Suivant la façon dont on conçoit le temps (linéaire ou ramifié, continu ou discret...), on modifiera la relation d'accessibilité, ce qui à son tour modifiera les axiomes de la logique.

Par analogie avec la logique modale, on définit également les opérateurs duaux de F et P :

1. $Gp = \neg F \neg p$
2. $Hp = \neg P \neg p$

Les définitions formelles du langage de la logique temporelle, ainsi que les définitions de satisfiabilité et de validité sont très semblables à celles de la logique modale, nous ne les reprendrons pas. Nous allons plutôt nous intéresser à l'influence des diverses interprétations du temps sur l'axiomatique (théorie de la démonstration) et les relations de satisfiabilité (théorie des modèles).

Notons tout de suite que deux axiomes seront valables dans toute la suite ; il s'agit de :

1. $G(A \rightarrow B) \rightarrow (GA \rightarrow GB)$
2. $H(A \rightarrow B) \rightarrow (HA \rightarrow HB)$

Ces axiomes, sont semblables à $\Diamond(A \rightarrow B) \rightarrow (\Diamond A \rightarrow \Diamond B)$ et $\Box(A \rightarrow B) \rightarrow (\Box A \rightarrow \Box B)$ en logique modale.

8.10.3 L'ultériorité

Il s'agit là de la notion de base dans la représentation intuitive que nous avons du temps. Elle impose (évidemment) à la relation entre les mondes d'être anti-symétrique. Les axiomes logiques de tout système temporel doivent donc avoir des modèles dont la relation d'accessibilité est anti-symétrique. Cependant, il n'existe pas d'axiomes caractérisant l'anti-symétrie. Les axiomes suivants, qui expriment l'ultériorité, correspondent bien à une relation anti-symétrique, mais ne correspondent pas seulement à une telle relation :

- $A \rightarrow GPA$: on peut le lire : « si A est vrai maintenant, alors il n'y aura pas de futur où A n'aura pas été vrai », c'est-à-dire que « si A est vrai maintenant, dans tous les états futurs A a été vrai ».
- $A \rightarrow HFA$: on peut le lire « si A est vrai maintenant, alors dans tous les états passés A sera vrai dans le futur ».

Comme nous le voyons ces axiomes se « lisent » bien en terme d'ultériorité comme nous le disions en tête de ce paragraphe. Avec les deux axiomes du paragraphe précédent, ils forment l'axiomatique de la logique temporelle de base.

8.10.4 Ordre partiel des dates

Si nous supposons que nous pouvons ordonner, tout au moins partiellement, nos dates, notre relation devient une relation d'ordre partiel *i.e.*, elle devient transitive en plus de son anti-symétrie.

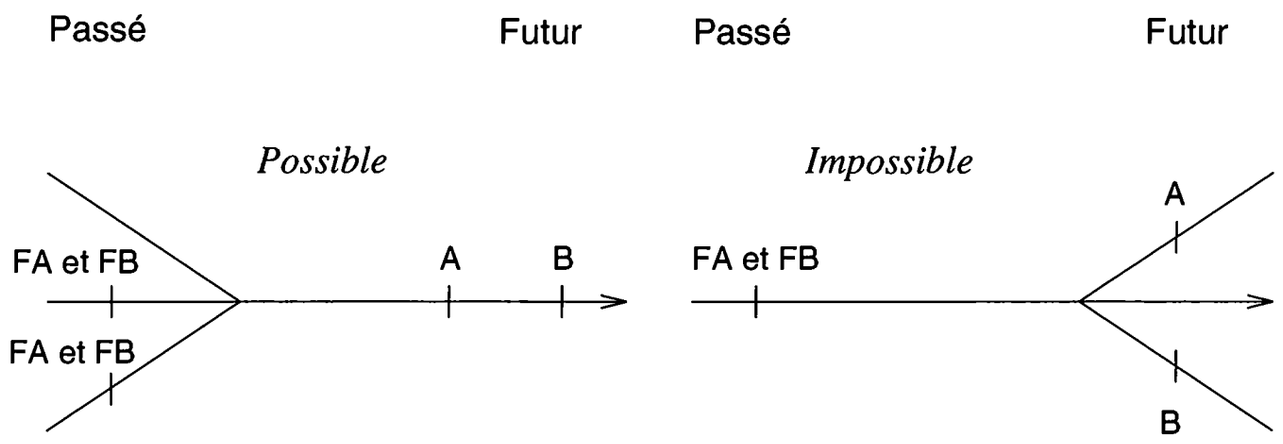


Figure 8.3 – Ordre total des dates futures

La transitivité exprime des propriétés comme : « si A est vrai dans le futur, alors il sera toujours vrai que A sera vrai ».

Il nous faut donc adjoindre des axiomes exprimant la transitivité. La transitivité est associée à l'axiome $\Box A \rightarrow \Box\Box A$. Cet axiome devient en logique temporelle $GA \rightarrow GGA$.

8.10.5 Ordre total des dates

Nous conservons les propriétés (anti-symétrie et transitivité) précédentes, ainsi que les axiomes qui en découlent, mais nous devons de plus exprimer que deux dates sont toujours comparables : $\forall w_i \forall w_j (w_i < w_j) \vee (w_j < w_i) \vee (w_i = w_j)$.

Les axiomes correspondant à cette propriété sont :

- $FA \wedge FB \rightarrow F(A \wedge FB) \vee F(FA \wedge B)$ qui indique l'ordre total des dates futures.
- $PA \wedge PB \rightarrow P(A \wedge PB) \vee P(PA \wedge B)$ qui indique l'ordre total des dates passées.

L'axiome $FA \wedge FB \rightarrow F(A \wedge FB) \vee F(FA \wedge B)$ exprime que si à un instant donné nous avons dans le futur deux événements A et B, alors soit A se produit et B sera dans le futur de A, soit B se produit et A sera dans le futur de B. Ceci nous donne le schéma de la figure 8.3. Pour plus de détails, on pourra se reporter à (Rescher and Urquhart 1971).

Partie II

Éléments d'informatique théorique

Se demander si un ordinateur sait penser est aussi intéressant que se demander si un sous-marin sait nager.

— *E. W. Dijkstra*

CHAPITRE 9

Théorie des langages formels

Jean-Marc Alliot

9.1 Introduction

La théorie des langages formels forme une des parties les plus importantes de l'informatique d'aujourd'hui. Comme il s'agit, d'autre part, d'un domaine pluri-disciplinaire, étudié à la fois par les mathématiciens, les informaticiens, les linguistes, les psychologues et jusqu'aux biologistes, il est indispensable d'en exposer les éléments de base.

Les principales applications des langages formels sont :
- l'étude de la calculabilité et de la complexité ;
- la compilation ;
- l'étude des langues naturelles ;
- l'étude des systèmes formels en mathématiques[1] ;
- les automates.

Il est impossible, dans un chapitre d'une dizaine de pages, de présenter l'intégralité de la théorie des langages formels qui réclamerait plusieurs ouvrages. Nous nous contenterons d'en présenter les points principaux. De même, nombre de théorèmes seront énoncés sans démonstration. On peut se référer à (Salomaa 1989) pour une présentation plus détaillée[2].

9.2 Définitions générales

Définition 9.1 – Alphabet, symboles – *Un alphabet est un ensemble fini non vide que nous noterons* Σ. *Les éléments de* Σ *sont appelés symboles, ou lettres.*

Définition 9.2 – Mot sur un alphabet Σ – *Un mot sur un alphabet* Σ *est une suite de* n *symboles de* Σ, *avec* $n \geq 0$. *Le mot composé de 0 symbole est appelé le mot vide et est noté* λ.

Définition 9.3 – Concaténation de mots – *L'opération de concaténation consiste à placer deux mots* m *et* m' *côte à côte pour former le mot noté* mm' *constitué de la suite des symboles*

1 Un système formel est très proche d'un système de réécriture de Post, comme nous allons le voir dans ce chapitre.

2 Nous conseillons également, pour une introduction aux fondements de l'informatique théorique, (Nivat 1984).

de m suivie de la suite des symboles de m'. On notera m^i le résultat de la concaténation de i mots, tous égaux à m.

Définition 9.4 – Langage engendré par un alphabet Σ – *On appelle langage engendré par l'alphabet Σ l'ensemble de tous les mots sur Σ. On notera cet ensemble Σ^*. On notera Σ^+ l'ensemble Σ^* privé du mot nul λ.*

Définition 9.5 – Langages formels sur un alphabet Σ – *Les langages formels sur l'alphabet Σ sont les sous-ensembles de Σ^*. (Souvent, on parlera simplement de langages sur Σ.)*

Définition 9.6 – Alphabet d'un langage formel L – *L'alphabet d'un langage L, noté $\alpha(L)$, est le plus petit ensemble Σ tel que L est un langage sur Σ.*

Définition 9.7 – Morphisme de langages – *Soit Σ et Δ deux alphabets. Un morphisme de langage f est une application définie de Σ^* dans Δ^* vérifiant la propriété : $\forall m \in \Sigma^*, \forall m' \in \Sigma^*, f(mm') = f(m)f(m')$*

Il est temps de donner quelques exemples. Prenons comme alphabet $\Sigma = \{a, b\}$. Alors Σ^* est l'ensemble des mots ne contenant que des a et des b. Le langage $L = \{ab, b, aba, bba, bbbbaba\}$ est un langage sur Σ, fini. Le langage $L' = \{a^p b^q, p \geq 0, q \geq 0\}$ est un autre langage sur Σ, infini.

L'alphabet du langage $L = \{a, ba\}$ est $\Sigma = \{a, b\}$. L'alphabet du langage $L' = \{a, aaaa, aa\}$ est $\{a\}$. On peut cependant remarquer que L' est aussi un langage sur Σ.

La concaténation des mots $m = aabaab$ et $m' = abb$ donne le mot $mm' = aabaababb$.

Notons enfin que, de par sa définition, un morphisme f de Σ^* sur Δ^* est entièrement déterminé par la liste de ses valeurs sur l'alphabet Σ, noté $f(\Sigma)$.

Avant d'aller plus loin, il faut signaler une propriété de dualité particulièrement importante dans l'ensemble des systèmes de transformation que nous allons étudier. En fait, tout système de transformation peut être considéré de deux façons : soit comme un système génératif, fabriquant des mots terminaux à partir de mots initiaux ; soit comme un système de reconnaissance, qui, lui, sera apte à reconnaître des mots appartenant à un langage. Cependant certains systèmes (les grammaires, les systèmes de Post) sont plus adaptés au fonctionnement génératif alors que les algorithmes de Markov sont eux plus adaptés à des mécanismes de reconnaissance. Nous étudierons chacun de ces systèmes dans sa forme la plus classique, mais nous devons garder en mémoire l'aspect dual du mécanisme.

9.3 Systèmes de réécriture

Les systèmes de réécriture permettent de construire des langages à partir d'un alphabet et de règles de formation des mots du langage, appelées règles de production. Il existe plusieurs façons d'utiliser les systèmes de réécriture pour construire des langages. Dans ce paragraphe nous nous contenterons de présenter le mécanisme formel de fonctionnement des systèmes de réécriture. Notons que les systèmes de réécriture ne permettent de formaliser que des systèmes où l'on ne change qu'une partie du mot à la fois. Nous

verrons qu'il existe des méthodes de réécriture qui modifient simultanément toutes les parties du mot qu'il est possible de réécrire.

Définition 9.8 – Système de réécriture – *Un système de réécriture est un triplet* $\mathcal{R} = (\Sigma, \mathcal{P}, \rightsquigarrow)$ *où* Σ *est un alphabet,* $\mathcal{P}$ *un ensemble fini de couples (m,m'), où m et m' sont des mots sur* Σ*, et* $\rightsquigarrow$ *est la relation de production directe.*

Les couples (m,m') sont appelés règles de production, on les note également $m \rightsquigarrow m'$.

La relation de production directe de $\mathcal{R}$*, notée* $\rightsquigarrow$ *est définie par :* $x \rightsquigarrow y$ *ssi il existe des mots* x_1, x_2 *et une règle de production* $m \rightsquigarrow m'$ *tels que* $x = x_1 m x_2$ *et* $y = x_1 m' x_2$.

On dit alors que y est produit directement par x en accord avec $\mathcal{R}$.

Définition 9.9 – Dérivation – *On appelle relation de dérivation la relation notée* $\rightsquigarrow_*$ *définie par :* $x \rightsquigarrow_* y$ *ssi il existe une suite de mots* $m_1, m_2, \ldots, m_p$ *avec* $p \geq 0$ *telle que* $x \rightsquigarrow m_1 \rightsquigarrow m_2 \rightsquigarrow \ldots \rightsquigarrow m_p \rightsquigarrow y$.

On dit alors que y dérive[3] *de x. (Notons que p peut valoir 0, dans ce cas la suite de mots* m_i *est vide).*

9.4 Grammaires

Les grammaires sont les premières applications des systèmes de réécriture que nous allons voir. Le but d'une grammaire est la production d'un langage à partir d'un alphabet et de règles de production.

9.4.1 Éléments de base

Définition 9.10 – Grammaire – *Une grammaire est définie par un quadruplet* $G = (\mathcal{R}, \Sigma_N, \Sigma_T, D)$. $\mathcal{R} = (\Sigma, \mathcal{P}, \rightsquigarrow)$ *est un système de réécriture,* Σ_N *et* Σ_T *sont deux alphabets tels que* $\Sigma = \Sigma_N \cup \Sigma_T$, D *est une lettre de* Σ_N.

Définition 9.11 – Langage généré par une grammaire – *Le langage généré par la grammaire G est défini par :* $L(G) = \{m \in \Sigma_T^* \mid D \rightsquigarrow_* m\}$

Les symboles de Σ_N sont appelés symboles non-terminaux du langage, les symboles de Σ_T sont les symboles terminaux. D est la lettre de départ. Notons qu'il est de tradition de noter les éléments de Σ_N par des lettres majuscules et les éléments de Σ_T par des lettres minuscules.

Définition 9.12 – Grammaires équivalentes – *Deux grammaires sont dites équivalentes si elles génèrent le même langage.*

Définition 9.13 – Langage récursivement énumérable – *Un langage L est dit récursivement énumérable ssi il existe une grammaire G telle que* $L = L(G)$.

La définition ci-dessus est la définition donnée par la plupart des auteurs. Disons cependant que nous pourrions aussi parler de langage Turing-calculable. En effet, un

---

3 Certains auteurs disent aussi que x produit y. Nous avons renoncé à cette terminologie pour éviter les confusions avec la relation de production directe.

langage est récursivement énumérable si et seulement si il existe une machine de Turing permettant de reconnaître l'ensemble de ses mots *i.e.*, de vérifier si un mot est effectivement un mot du langage[4]. Il est clair que si un langage est généré par une grammaire, il est simple de construire une machine de Turing simulant le comportement de cette grammaire. Réciproquement, si nous connaissons une machine de Turing permettant d'accepter chaque mot du langage, nous saurons construire un système de réécriture et une grammaire générant ce langage à partir de l'alphabet de notre machine de Turing et de son programme.

Définition 9.14 – Grammaire et langage algébrique –

Une grammaire est dite algébrique (ou libre de contexte, de l'anglais context free*) ssi toutes les règles du système de production sont de la forme* $A \rightsquigarrow m$ *où* A *est une lettre de* Σ_N *et* m *un mot quelconque* Σ^*.

Un langage est algébrique s'il est généré par une grammaire algébrique.

Intuitivement, le terme libre de contexte signifie que tout symbole non terminal peut être remplacé par une séquence quelle que soit sa position par rapport à d'autres lettres (son contexte). Ainsi, si une des règles de production était $aA \rightsquigarrow ab$, le symbole A ne pourrait être substitué par b que s'il était précédé de a. Ce type de grammaire n'est pas libre de contexte.

Définition 9.15 – Grammaire et langage réguliers – *Une grammaire est dite régulière ssi toutes les règles de production du système de réécriture sont de la forme* $A \rightsquigarrow m$ *ou* $A \rightsquigarrow mB$, *avec* A *et* B *des symboles de* Σ_N *et* m *un mot de* Σ_T^*.

Un langage est dit régulier s'il est engendré par une grammaire régulière

Remarquons que toute grammaire régulière est algébrique. Notons aussi que tout langage fini est régulier.

9.4.2 Exemples

Soit la grammaire G_1 définie par
- $\Sigma_N = \{D\}$
- $\Sigma_T = \{a, b\}$
- Les règles de production du système de réécriture sont :
  1. $D \rightsquigarrow aD$
  2. $D \rightsquigarrow Db$
  3. $D \rightsquigarrow \lambda$

La grammaire G_1 est algébrique, mais n'est pas régulière à cause de la règle 2. Un exemple de dérivation en accord avec G_1 serait par exemple :

$$D \rightsquigarrow aD \rightsquigarrow aaD \rightsquigarrow aaDb \rightsquigarrow aab$$

4 Le problème d'appartenance est alors semi-décidable : si le mot appartient au langage L, nous pourrons le reconnaître en un temps fini. Ce n'est pas forcément le cas si il n'appartient pas à L. Un langage sera dit *récursif* si lui et son complémentaire dans Σ^* sont *récursivement énumérables*. Dans ce cas, le problème de reconnaissance d'un mot quelconque est effectivement décidable.

Il est clair que $L(G_1) = \{a^p b^q, p \geq 0, q \geq 0\}$

Il est possible de construire des grammaires plus complexes ; soit G_2 définie par :

- $\Sigma_N = \{D\}$
- $\Sigma_T = \{a, b\}$
- Les règles de production du système de réécriture sont :
  1. $D \rightsquigarrow DD$
  2. $D \rightsquigarrow ba$
  3. $aD \rightsquigarrow ab$

Cette grammaire n'est pas algébrique du fait de la règle 3.

9.4.3 Formes normales

Nous avons vu que des grammaires différentes peuvent engendrer le même langage. La question que l'on peut alors se poser est la suivante : existe-t-il pour un langage donné une grammaire « plus simple » que les autres qui l'engendre ?

Le critère de « plus simple » est difficile à définir, mais on peut considérer que les trois théorèmes suivants fournissent une forme de réponse à cette question.

Dans les règles de production citées dans les théorèmes suivants, les lettres majuscules représentent toujours des symboles non-terminaux et les lettres minuscules des symboles terminaux.

Théorème 9.1 – Forme normale des grammaires régulières – *Soit L un langage régulier. Alors L est généré par une grammaire telle que chaque règle est soit du type $A \rightsquigarrow aB$, soit du type $A \rightsquigarrow \lambda$.*

Nous ne donnerons pas la démonstration de ce théorème. Elle consiste principalement à supprimer dans la grammaire initiale qui génère le langage, toutes les règles de production qui n'ont pas une forme correcte, en ajoutant des symboles non-terminaux dans Σ_N pour conserver la même puissance génératrice au langage.

Théorème 9.2 – Forme normale de Chomsky – *Soit L un langage algébrique, tel que $\lambda \notin L$. Alors il existe une grammaire G générant L telle que toutes les règles de production de G sont du type $A \rightsquigarrow BC$ ou $A \rightsquigarrow a$.*

La démonstration est similaire dans son principe à la précédente. Signalons qu'en rajoutant la règle de production $A \rightsquigarrow \lambda$ nous sommes en mesure de générer tous les langages algébriques.

Théorème 9.3 – Forme normale d'une grammaire – *Soit L un langage récursivement énumérable. Alors il existe une grammaire G générant L et telle que toutes les règles de production de G sont du type : $A \rightsquigarrow BC$, $AB \rightsquigarrow CD$, $A \rightsquigarrow a$ ou $A \rightsquigarrow \lambda$.*

Ce résultat fournit donc une forme normale pour toute grammaire, au sens de la capacité génératrice de cette grammaire.

9.5 Les systèmes de Post

Nous présentons les systèmes de Post pour montrer à quel point l'informatique théorique est liée à la logique formelle. En effet, les systèmes de Post furent utilisés pour la

construction des systèmes formels en logique. Nous aurions pu d'ailleurs présenter une axiomatique de $\mathbb{N}$ basée sur les systèmes de Post. Le principe des systèmes de Post est de fournir un ensemble de mots, qui forment les axiomes, des ensembles de variables et de constantes et des règles de production : exactement ce qui est nécessaire à une axiomatique formelle.

Définition 9.16 – Système de Post – *Un système de Post est un quadruplet* $\mathcal{PS} = (\Sigma_v, \Sigma_c, \mathcal{P}, \mathcal{A})$. Σ_v *est l'alphabet des variables et* Σ_c *l'alphabet des constantes, vérifiant* $\Sigma_v \cap \Sigma_c = \varnothing$. $\mathcal{A}$ *est un ensemble fini de mots sur* Σ_c*, appelés axiomes.* $\mathcal{P}$ *est un ensemble de règles de production de la forme* $(\alpha_1, \alpha_2, \ldots, \alpha_k \rightsquigarrow \alpha)$ *où les* α_i *et* α *sont des mots sur* $\Sigma_v \cup \Sigma_c$.

Il nous reste à mettre en place une des opérations fondamentales des systèmes de Post : le mécanisme de substitution.

Définition 9.17 – Substitution uniforme – *Soit un système de Post* $\mathcal{PS}$ *et une règle de production* $R = (\alpha_1, \alpha_2, \ldots, \alpha_k \rightsquigarrow \alpha)$ *de* $\mathcal{PS}$. *On appelle* substitution uniforme *l'opération consistant à remplacer dans* R *toutes les occurrences des symboles appartenant à* Σ_v *par des mots de* Σ_c^**, chaque occurrence d'un symbole de* Σ_v *étant remplacé par le même mot de* Σ_c^*.

La nouvelle règle de production ainsi obtenue après substitution uniforme est appelée version constante *de* R *et nous la noterons* $R_c = (\beta_1, \beta_2, \ldots, \beta_k \rightsquigarrow \beta)$.

Définition 9.18 – Théorème d'un système de Post – *Un mot* m *de* Σ_c *est un théorème d'un système de Post* $\mathcal{PS}$ *ssi :*

- m *est un axiome* ($m \in \mathcal{A}$),
- *il existe une version constante* $R_c = (\beta_1, \beta_2, \ldots, \beta_k \rightsquigarrow \beta)$ *d'une règle de production* R *de* $\mathcal{P}$ *telle que* $\beta_1, \beta_2, \ldots, \beta_k$ *sont des théorèmes et* $m = \beta$.

On retrouve la définition (informelle) que nous avions donnée des théorèmes et des axiomes dans les chapitres précédents. Les règles d'inférence sont devenues les règles de production du système de Post.

Définition 9.19 – Langage généré – *On appelle langage généré par un système de Post* $\mathcal{PS}$ *et un alphabet* Σ_T*, avec* $\Sigma_T \subset \Sigma_c$ *l'ensemble :*

$$L(\mathcal{PS}, \Sigma_T) = \{m \in \Sigma_T^* \mid m \text{ est un théorème de } \mathcal{PS}\}$$

Si $\Sigma_T = \Sigma_c$*, on dit que le langage est généré par un* système de Post pur.

Définition 9.20 – Système de Post régulier – *Un système de Post est dit régulier si toutes les règles de production sont de la forme* $mB \rightsquigarrow m'B$ *où* m *et* m' *sont des mots de* Σ_C^* *et* B *est une variable.*

Nous allons donner un exemple clarifiant les définitions précédentes.

9.5.1 Une axiomatique du calcul propositionnel

Le système de Post suivant offre une nouvelle formalisation du calcul propositionnel :

- $\Sigma_v = \{A, B, C, \ldots\}$
- $\Sigma_c = \{\rightarrow, \neg, a, b, c, \ldots\}$
- $\mathcal{P} = \{((A, A \rightarrow B) \rightsquigarrow B)\}$

– l'ensemble $\mathcal{A}$ contient les trois mots suivants :

1. $A \rightarrow (B \rightarrow A)$
2. $((A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C)))$
3. $((\neg B \rightarrow \neg A) \rightarrow ((\neg B \rightarrow A) \rightarrow B))$

Nous voyons ainsi qu'il est parfaitement possible de réaliser une axiomatique formelle basée uniquement sur la théorie des langages.

9.5.2 Propriétés

Théorème 9.4 – Équivalence systèmes de Post—grammaires – *L est un langage récursivement énumérable ssi il existe un système de Post $\mathcal{PS}$ et un alphabet Σ_T tels que $L = L(\mathcal{PS}, \Sigma_T)$.*

Donc la puissance génératrice des systèmes de Post est équivalente à celle des grammaires.

Théorème 9.5 – Forme normale d'un système de Post – *Soit L un langage généré par un système de Post. Alors il existe un alphabet Σ_T et un système de Post $\mathcal{PS}$ tels que $L = L(\mathcal{PS}, \Sigma_T)$ et dont toutes les règles de production sont de la forme $uA \rightsquigarrow Aw$ avec $u, w \in \Sigma_c^*$ où A une variable.*

Ce théorème est très semblable dans son principe à ceux que nous avions établis au paragraphe précédent pour les grammaires.

Théorème 9.6 – *Soit L un langage régulier. Alors il existe un système de Post $\mathcal{PS}$ pur et régulier tel que $L = L(\mathcal{PS}, \Sigma_c)$.*

Réciproquement, tout langage engendré par un système de Post pur et régulier est régulier.

Ce théorème fournit une relation intéressante entre grammaire régulière et système de Post. Il est assez difficile à démontrer. Nous renvoyons là-encore le lecteur intéressé à (Salomaa 1989).

9.6 Algorithme de Markov

Cette technique de réécriture présente une caractéristique intéressante, comparée aux deux méthodes que nous venons d'examiner : elle est déterministe. À chaque instant, une et une seule règle de production est applicable, alors que dans les grammaires et les systèmes de Post, on a généralement le choix entre plusieurs possibilités à chaque étape du calcul.

Les algorithmes de Markov sont donc plus particulièrement adaptés à la reconnaissance de mot qu'à la génération de langage.

Définition 9.21 – Algorithme de Markov – *Un algorithme de Markov est un triplet $\mathcal{MA} = (\Sigma, \mathcal{P}, \mathcal{T})$ où Σ est un alphabet, $\mathcal{P}$ une suite ordonnée de règles de production de la forme : $\alpha_1 \rightsquigarrow \beta_1, \alpha_2 \rightsquigarrow \beta_2, \ldots \alpha_p \rightsquigarrow \beta_p$, $\mathcal{T}$ est un sous-ensemble de $\mathcal{P}$ appelé ensemble des règles de production terminales, les α_i, β_i sont des mots de Σ^*.*

On parle d'*algorithme* de Markov car on peut décrire la transformation d'un mot m en un mot m' sous la forme de l'algorithme suivant. On commence par prendre un mot x avec $x = m$ et on applique les opérations :

1. on recherche dans le mot x la première occurrence d'un des mots α_i *en commençant par le début* de l'énumération des α_i. Si on ne trouve aucune occurrence, la transformation est terminée et $m' = x$;
2. on remplace dans le mot x la première occurrence du mot α_i par le mot β_i. Si la règle de production $\alpha_i \rightsquigarrow \beta_i$ est une règle de $\mathcal{T}$, alors la transformation est terminée et $m' = x$;
3. sinon on reprend à l'étape 1.

Si l'algorithme de Markov $\mathcal{MA}$ se termine pour le mot m et donne le mot m', on dit que $\mathcal{MA}$ *transforme* m en m' et l'on notera $m \rightsquigarrow m'$[5]. Si l'algorithme ne se termine pas, on dit que $\mathcal{MA}$ boucle pour le mot m.

Comme nous le voyons, un algorithme de Markov ne peut générer, à partir d'un ensemble d'axiomes fini, qu'un ensemble fini de mots, puisque chaque axiome ne peut produire qu'un seul mot. Pour cette raison, les algorithmes de Markov sont surtout utilisés pour reconnaître des mots *i.e.*, vérifier s'ils appartiennent à un langage.

Définition 9.22 – Langage reconnu par un algorithme de Markov – *Soit* $\mathcal{MA} = (\Sigma, \mathcal{P}, \mathcal{T})$ *un algorithme de Markov et* $\Sigma_T \subset \Sigma$ *un alphabet. Le langage reconnu par* $\mathcal{MA}$ *respectant* Σ_T *est défini par :*

$$L(\mathcal{MA}, \Sigma_T) = \{m \in \Sigma_T^* \mid m \rightsquigarrow \lambda\}$$

Prenons comme exemple le système de Markov et l'alphabet Σ_T définis par :
- $\Sigma = \{a, b, S\}$
- $\Sigma_T = \{a, b\}$
- $\mathcal{P} = \{ab \rightsquigarrow S, aSb \rightsquigarrow S, S \rightsquigarrow \lambda\}$
- $\mathcal{T} = \{S \rightsquigarrow \lambda\}$

Cet algorithme de Markov reconnaît le langage : $L = \{a^i b^i, i \geq 0\}$

Signalons le résultat fondamental suivant :

Théorème 9.7 – *Un langage est récursivement énumérable ssi il est de la forme* $L(\mathcal{MA}, \Sigma_T)$.

Ce théorème énonce que les langages reconnus par les algorithmes de Markov sont exactement les langages générables par des grammaires ou des systèmes de Post.

En fait, tous ces résultats étaient prévisibles. Les algorithmes de Markov sont très proches des machines de Turing et les systèmes de Post très proches des systèmes formels. Nous retrouvons donc des parentés fort naturelles.

---

5 Attention, d'autres auteurs réservent cette notation à la relation de production directe, que nous n'avons pas définie pour les algorithmes de Markov.

CHAPITRE 10

La calculabilité

Jean-Marc Alliot

10.1 Introduction

Nous avons introduit les notions fondamentales de calculabilité et de décidabilité dans le chapitre consacré aux systèmes formels. Nous n'y reviendrons pas.

D'autre part, la démonstration d'indécidabilité pour un problème A se fait le plus souvent par référence à un problème B dont on a déjà montré l'indécidabilité : on montre que si le problème A était décidable, alors le problème B le serait aussi. Nous considérons comme acquis le résultat démontré dans le chapitre 5 consacré aux systèmes formels *i.e.*, le prédicat $P_T = \exists n T(i, i, n)$ est indécidable, ou encore le comportement d'une machine de Turing avec comme argument son index est indécidable. Nous nous servirons de ce résultat pour établir notre deuxième théorème d'indécidabilité.

Nous ne parlerons dans ce chapitre que de quelques problèmes de décision classiques. Il aurait sans doute aussi fallu, pour être plus général, parler des problèmes de Thue. Nous ne l'avons pas fait. Le lecteur intéressé peut se reporter à (Salomaa 1989).

10.2 Problème de reconnaissance

On appelle problème de reconnaissance d'un mot m, pour un langage L donné (ici récursivement énumérable), le problème consistant à déterminer si ce mot appartient au langage L. Nous avons vu qu'un langage L est récursivement énumérable ssi il existe une machine de Turing permettant précisément de reconnaître chaque mot de L. Donc si le problème de la reconnaissance était décidable, cela signifierait que le prédicat P_T serait décidable, ce qui est faux. Donc le problème de reconnaissance est indécidable.

Théorème 10.1 – Indécidabilité du problème de reconnaissance – *Le problème consistant à déterminer si un mot m donné appartient au langage généré par une grammaire G donnée est indécidable.*

Il est intéressant de bien réfléchir à ce théorème ; il signifie qu'il est impossible de trouver un algorithme qui garantisse de déterminer en un temps fini, et dans tous les cas, si un mot m appartient à un langage $L(G)$ généré par une grammaire G (ou par un système de Post). Il est évident qu'il existe des grammaires G pour lesquelles il est possible

de trouver un tel algorithme, de même qu'il existe des mots pour lesquels le problème de l'appartenance est trivialement reconnu, mais l'algorithme général de reconnaissance *n'existe pas.*

10.3 Le problème de correspondance de Post

Nous allons nous intéresser de façon plus attentive au problème de correspondance de Post, car il fournit un exemple simple, très étudié, d'un problème indécidable. Nous noterons souvent le problème de correspondance de Post en abrégé PCP.

10.3.1 Énoncé intuitif du problème

Nous allons tout d'abord présenter le problème de Post de façon informelle.

Considérons les deux listes suivantes :

$$(aabb, ab, aab)(bb, abaa, b)$$

Supposons que je concatène les mots de ma première liste dans l'ordre (1,3,2,1). J'obtiens le mot : $m_1 = aabbaababaabb$. Si j'applique la même liste d'indices sur ma seconde liste, j'obtiens le mot $m_2 = bbbabaabb$. Les mots m_1 et m_2 sont différents. Mais est-il possible de construire une liste d'indices telle que $m_1 = m_2$? Cette question est le problème de correspondance de Post, appliqué ici aux listes (abb, ab, aba) et (bb, aba, ba). Dans le cas présent, cette *instance* du PCP a une solution, il suffit de prendre comme liste d'indices (2,3,2,1) qui nous donne : $abaababaabb$.

La question de la décidabilité du problème de correspondance de Post est donc : peut-on trouver un algorithme général permettant de dire si le PCP a une solution pour deux listes quelconques, en un temps fini ?

10.3.2 Démonstration de l'indécidabilité

Nous allons montrer que le PCP est équivalent au problème de l'appartenance à un langage récursivement énumérable. Ce dernier problème étant indécidable, le problème de Post le sera également.

Pour ce faire, nous allons, pour une grammaire donnée G, construire une instance du problème de Post telle que la résolution de cette instance permettra de savoir si un mot m appartient à $L(G)$. Réciproquement, suivant que $m \in L(G)$ ou $m \notin L(G)$, le problème de Post aura, ou n'aura pas, de solution.

Avant de nous lancer dans la construction du PCP pour une grammaire quelconque G, nous allons formaliser le problème de correspondance.

Définition 10.1 – Problème de correspondance de Post – *Soit f et g deux morphismes définis de Σ^* dans Δ^*, avec Σ et Δ deux alphabets.*

Le couple (f, g) est une instance du problème de correspondance de Post. Un mot m de Σ^* est une solution pour cette instance ssi $f(m) = g(m)$.

Ainsi, si nous considérons l'exemple précédent, nous avons :

- $\Sigma = \{1, 2, 3\}$
- $\Delta = \{a, b\}$
- f défini par : $f(1) = abb$, $f(2) = ab$, $f(3) = aba$
- g défini par : $g(1) = bb$, $g(2) = aba$, $g(3) = ba$

La solution de cette instance du problème de Post est le mot de Σ^*, $m = 2321$.

Considérons maintenant une grammaire $G = (\mathcal{R}, \Sigma_N, \Sigma_T, D)$. Nous supposons que les règles de production de $\mathcal{R}$ sont telles que λ n'apparaît ni dans la partie droite ni dans la partie gauche des règles de production[1]. Les règles de production seront : $\{\alpha_i \rightarrow \beta_i, 0 < i \leq n\}$.

Posons que $\Gamma = \Sigma_N \cup \Sigma_T = \{a_1, a_2, \ldots, a_p\}$ et construisons un alphabet $\Lambda = \{a'_i \mid a_i \in \Gamma\}$.

Nous considérons alors l'alphabet $\Delta = \Gamma \cup \Lambda \cup \{I, F, \#, \$\}$. Cet alphabet sera l'alphabet d'arrivée de nos morphismes f et g. L'alphabet de départ sera : $\Sigma = \{1, 2, 3, \ldots, 2n + 2p + 4\}$. Nous notons également α'_i et β'_i les mots α_i et β_i dans lesquels on a substitué les lettres a'_i à toutes les occurrences des lettres a_i.

Nous disons que l'instance du PCP que nous venons de construire admet une solution ssi le mot m appartient à $L(G)$, les morphismes f et g étant définis par le tableau suivant :

	1	2	3	4	$4+i$	$4+p+i$	$4+2p+j$	$4+2p+n+j$
f	$Ia_1\#$	\$	#	F	a'_i	a_i	β'_j	β_j
g	I	#	\$	\$$mF$	a_i	a'_i	α_j	α'_j

La démonstration complète de ce théorème est relativement technique, le lecteur peut se reporter à (Salomaa 1973) pour plus de détails.

Intuitivement, les morphismes f et g simulent le comportement de la grammaire G. Le marqueur I désigne le début de la dérivation, le marqueur F, la fin de la dérivation, et les marqueurs # et \$ séparent les étapes des dérivations. L'astuce consiste à faire en sorte que le morphisme f démarre plus vite que le morphisme g et va effectivement générer le mot m, alors que le morphisme g ne pourra combler son retard que si le mot m est bien le résultat de la construction, en utilisant son indice 4.

Voyons tout cela sur un exemple. Soit la grammaire :

- $\Gamma = \{a_1 = D, a_2 = a, a_3 = b\}$, $(p = 3)$
- les règles de production sont : $D \rightarrow bDb$, $D \rightarrow aa$, $D \rightarrow aD$, $(n = 3)$

Le mot $baaab$ appartient à $L(G)$. On peut en effet l'obtenir par la dérivation : $D \rightsquigarrow bDb \rightsquigarrow baDb \rightsquigarrow baaab$.

Construisons l'instance du problème de Post associée à cette grammaire et à ce mot m :

- $\Delta = \{a_1 = D, a_2 = a, a_3 = b, a'_1 = D', a'_2 = a', a'_3 = b', I, F, \#, \$\}$
- $\Sigma = \{1, 2, 3, \ldots, 15, 16\}$

Nous pouvons alors construire le tableau des morphismes f et g :

---

1 Nous avons démontré dans le théorème 9.3 qu'il existait toujours une forme normale telle que λ n'apparaissait pas dans la partie gauche des règles de production. Il est également possible, moyennant quelques acrobaties techniques, de le faire disparaître de la partie droite, sans restreindre de façon significative le langage généré.

	1	2	3	4	5	6	7	8
f	$ID\#$	\$	#	F	D'	a'	b'	D
g	I	#	\$	$\$baaabF$	D	a	b	D'

	9	10	11	12	13	14	15	16
f	a	b	$b'D'b'$	$a'a'$	$a'D'$	bDb	aa	aD
g	a'	b'	D	D	D	D'	D'	D'

Il ne nous reste plus qu'à montrer comment nous construisons la solution du problème de correspondance de Post à partir de la dérivation de *baaab* mentionnée ci-dessus.

Le mécanisme est très simple. On reprend la dérivation, on place le marqueur de tête I (indice 1), puis on remplace chacun des $\rightsquigarrow$ par, alternativement, un # (occurrences impaires : première et troisième occurrence), ou un \$ (occurrence paire : deuxième occurrence).

Les pas pairs sont primés (*bDb*, *baaab*), les pas impairs (*D*, *baDb*) conservés inchangés ce qui nous donne :

$$ID\#b'D'b'\$baDb\#b'a'D'b'\$baaabF$$

On arrive alors, en suivant pas à pas le tableau des deux morphismes, à la construction du nombre : 1.11.2.10.16.10.3.7.6.5.7.2.10.9.15.10.4

Ce nombre est solution de l'instance du PCP considérée[2].

On reconnaît facilement dans ce mot la dérivation issue de la grammaire G.

Réciproquement, si le problème de Post a une solution, elle commence impérativement par l'indice 1, se termine par l'indice 4 et permet de reconstruire aisément la dérivation qui générera le mot m pour la grammaire G (nous ne le démontrerons pas).

On constate donc l'équivalence du problème de Post et du problème de reconnaissance pour les langages récursivement énumérables. D'où le résultat :

Théorème 10.2 – Indécidabilité du PCP – *Le problème de correspondance de Post est indécidable.*

10.3.3 Autres résultats sur le PCP

Le problème de correspondance de Post est un problème qui a été, et reste, très étudié.

On s'est posé deux questions principales sur le problème de Post :

1. quelle est l'influence du cardinal de l'alphabet de départ sur la décidabilité du problème ?
2. quelle est l'influence du cardinal de l'alphabet d'arrivée sur la décidabilité du problème ?

---

2 Le lecteur ébahi par ce tour de magie est invité à le recommencer lui-même sur un autre exemple afin de mieux comprendre les mécanismes sous-jacents.

La question (2) est relativement facile à résoudre : il est simple de prouver que si l'alphabet d'arrivée est réduit à une seule lettre, le problème est décidable. En effet, nous sommes ramenés à deux listes de la forme : $(a^{p_1}, a^{p_2}, \ldots, a^{p_n})$ et $(a^{q_1}, a^{q_2}, \ldots, a^{q_n})$.

On est donc ramené à un problème d'arithmétique élémentaire : déterminer si l'équation

$$\sum_{i=1}^{n} p_i x_i = \sum_{i=1}^{n} q_i x_i \tag{10.1}$$

a des solutions entières positives ou nulles, mais non toutes identiquement nulles. La démonstration se fait en trois points :

1. s'il existe un j tel que $p_j = q_j$ alors le problème a trivialement comme solution $x_j = 1$ et tous les autres x_i nuls ;
2. s'il existe un j et un k tels que $p_j > q_j$ et $p_k < q_k$ alors le problème admet comme solution $x_j = q_k - p_k$ et $x_k = p_j - q_j$, tous les autres x_i nuls ;
3. si aucune des deux conditions précédentes n'est vraie alors on a soit (1) $\forall i p_i > q_i$ ou (2) $\forall i p_i < q_i$. L'équation 10.1 étant équivalente à :

$$\sum_{i=1}^{n} (p_i - q_i) x_i = 0 \tag{10.2}$$

Dans le cas (1), $p_i - q_i$ est toujours strictement positif, et strictement négatif dans le cas (2). Donc l'équation admet comme unique solution la solution identiquement nulle. Donc le problème n'est pas soluble.

Le problème de Post ayant un alphabet d'arrivée réduit à une lettre est donc décidable.

Nous allons maintenant prouver que si l'alphabet d'arrivée comprend deux lettres, alors le problème de correspondance est indécidable. Soit une instance (f, g) du problème de Post dont l'alphabet d'arrivée est $\Delta = \{a_1, \ldots, a_p\}$. Codons les lettres de Δ par $h(a_i) = ba^i b$. Considérons maintenant la nouvelle instance du problème de Post (hf, hg). Il s'agit là d'un problème dont l'alphabet d'arrivée est $\{a, b\}$ et qui est équivalent au problème précédent. Donc le problème de Post est indécidable quand l'alphabet d'arrivée contient deux lettres ou plus, car s'il était décidable tout problème de Post le serait également (et nous venons de démontrer le contraire).

Théorème 10.3 – *Le problème de Post est décidable pour un alphabet de départ quelconque ssi l'alphabet d'arrivée est réduit à une lettre.*

Quelle est maintenant l'influence du cardinal de l'alphabet de départ ? Le problème est encore ouvert. On sait seulement qu'il existe un nombre n compris entre 3 et 9 tel que le problème de Post soit décidable lorsque le cardinal de l'alphabet de départ est inférieur strictement à n, et indécidable pour tout autre valeur.

10.3.4 Autres résultats liés au PCP

L'indécidabilité du PCP a une grande importance en théorie des langages. Nous citons ici quelques théorèmes qui en découlent.

Théorème 10.4 – *Il n'existe pas d'algorithme permettant de savoir si deux langages à contexte libre ont une intersection non vide.*

Théorème 10.5 – *Il n'existe pas d'algorithme permettant de déterminer si un langage à contexte libre donné est régulier.*

Théorème 10.6 – *Il n'existe pas d'algorithme permettant de déterminer si un langage à contexte libre donné est égal à un langage régulier donné.*

10.4 Les solutions d'équations diophantiennes

La notion d'équation diophantienne remonte à l'antiquité. Son nom vient du mathématicien grec Diophante. Les équations diophantiennes constituent un des grands problèmes encore ouverts de la théorie arithmétique moderne, alors qu'on les étudie depuis plus de 20 siècles.

10.4.1 Définitions

Définition 10.2 – Ensemble diophantien – *Un ensemble E de n-uplets $(x_1, \ldots, x_n)$ d'entiers positifs ou nuls est dit diophantien si et seulement s'il existe un polynôme $P(x_1, \ldots, x_n, y_1, \ldots, y_m)$ à coefficients entiers positifs ou négatifs tel que :*

$$(x_1, \ldots, x_n) \in E \Leftrightarrow \exists (y_1 \geq 0, \ldots, y_m \geq 0), P(x_1, \ldots, x_n, y_1, \ldots, y_m) = 0 \quad (10.3)$$

Les équations du type 10.3 sont appelées *équations diophantiennes.*

On parle également de *relations diophantiennes* pour désigner les ensembles du type E.

Définition 10.3 – Degré d'un ensemble diophantien – *Le degré d'un ensemble E diophantien est le nombre k tel que k est le plus petit des degrés des polynômes permettant de définir E.*

Définition 10.4 – Dimension d'un ensemble diophantien – *On appelle dimension d de l'ensemble diophantien E le plus petit nombre possible de variables m sur l'ensemble de polynômes P permettant de définir E.*

Un exemple d'ensemble diophantien est :

$$\{x \mid \exists (y, z), x = yz\}$$

En fait, comme nous le verrons plus loin, les équations diophantiennes recouvrent un large ensemble de problèmes mathématiques.

10.4.2 Résultats fondamentaux

Théorème 10.7 – Polynôme décrivant un ensemble diophantien – *Pour tout ensemble diophantien E d'entiers positifs, il existe un polynôme Q décrivant exactement E, c'est-à-dire tel que l'ensemble des valeurs positives de Q, lorsque les variables de Q décrivent $\mathbb{N}$, est exactement E.*

Par définition de E, il existe P tel que

$$x \in E \Leftrightarrow \exists(y_1, \ldots, y_m), P(x, y_1, \ldots, y_m) = 0$$

Il nous suffit alors de poser :

$$Q(x, y_1, \ldots, y_m) = x(1 - (P(x, y_1, \ldots, y_m))^2)$$

Le résultat est évident si l'on a bien en mémoire que les polynômes P et Q ne prennent que des valeurs entières.

Théorème 10.8 – Degré d'un ensemble diophantien – *Le degré d'un ensemble diophantien est inférieur ou égal à 4.*

La démonstration de ce théorème est relativement simple. Elle réside dans l'introduction de variables supplémentaires, que nous utilisons chaque fois qu'une variable v_i apparaît dans P avec un degré supérieur à 2.

On pose ainsi $z_i = v_i^p$. Les termes de la forme v_i^{2p+1} deviennent $v_i z_i^2$ et les termes v_i^{2p} deviennent z_i^2. La relation initiale $P(v_k) = 0$ est alors remplacée par un ensemble de relations :

$$P'(v_k, z_l) = 0, P_i = (z_i - v_i^p) = 0$$

Le polynôme P' est de degré inférieur ou égal à 2 et il est possible d'appliquer la méthode précédente sur tous les polynômes P_i de degré plus grand que 2 pour se ramener finalement à un ensemble de relations de la forme :

$$R_1 = 0, R_2 = 0, \ldots, R_p = 0$$

où les R_i sont tous des polynômes de degré inférieur ou égal à 2. Le polynôme :

$$P >>= R_1^2 + R_2^2 + \cdots + R_p^2$$

est un polynôme de degré inférieur ou égal à 4 qui représente le même ensemble que notre polynôme P original. On remarquera que l'opération a finalement consisté à augmenter la dimension de P pour diminuer son degré.

Il faut savoir qu'un théorème presque similaire existe concernant la dimension *i.e.*, on sait qu'il existe un nombre n tel que tout ensemble diophantien est de dimension inférieure à n. On ne connaît pas encore la valeur de n.

Nous en venons maintenant au théorème fondamental concernant les ensembles diophantiens, qui va nous permettre de les mettre en rapport avec la théorie des langages formels et des machines de Turing :

Théorème 10.9 – *Tout ensemble récursivement énumérable est diophantien.*

Nous ne donnerons pas la preuve de ce théorème. On peut trouver des éléments plus élaborés concernant les ensembles diophantiens ainsi que la preuve de ce théorème dans (Davis 1982a) et (Davis 1973).

10.4.3 Le dixième problème de Hilbert

Lors de l'énoncé de l'ensemble de son programme, dont nous avons déjà parlé dans le chapitre consacré aux systèmes formels, le mathématicien David Hilbert énonça le problème suivant : *Trouver un algorithme permettant de déterminer si une équation polynomiale à coefficients entiers admet une solution entière.*

Ce problème resta ouvert jusqu'en 1973 ; le mathématicien américain Davis prouva que ce problème est indécidable. La démonstration de Davis était assez attendue. Les développements de la théorie des équations diophantiennes, ainsi que la théorie des fonctions récursives générales permettaient d'espérer un tel résultat. Nous allons simplement donner une idée de la démonstration d'indécidabilité.

Cette démonstration s'appuie sur le théorème 10.9. Elle se déroule en deux parties :

- on démontre tout d'abord que les ensembles de la forme :

$$S_i = \{x \mid \exists(y_1, y_2, \ldots, y_m), P_i(x, y_1, y_2, \ldots, y_m) = 0\}$$

  sont récursivement énumérables ;
- on en déduit qu'il existe un polynôme P_u appelé polynôme universel[3], tel que $x \in S_i$ ssi

$$\exists(y_1, y_2, \ldots, y_m), P_u(i, x, y_1, y_2, \ldots, y_m) = 0$$

- On remarque alors que pour un i_0 particulier il est impossible de savoir si l'équation $P_{i_0} = 0$ possède des solutions positives ou nulles, car si ce problème était décidable, le problème de la reconnaissance des langages récursivement énumérables serait décidable. Ce polynôme P_{i_0} montre que le dixième problème de Hilbert est indécidable.

Il existe quelques résultats supplémentaires intéressants sur les équations diophantiennes. On sait par exemple que tout problème diophantien de degré inférieur ou égal à 2 est décidable, et on sait également que les problèmes de degré 4 sont indécidables. On ne sait encore rien sur les problèmes de degré 3.

10.4.4 Conséquences

L'indécidabilité du problème de Hilbert est un résultat plus important qu'il n'y paraît.

On sait en effet ramener la plupart des problèmes encore ouverts en arithmétique à des problèmes diophantiens. Ainsi l'ensemble des nombres premiers est entièrement décrit par les valeurs positives du polynôme (Jones, Sato, Wada, Wiens) :

$$\begin{aligned}
&Q(a, b, \ldots, x, y, z) = \\
&\quad (k+2)(1 - (wz + h + j - q)^2 \\
&\quad - ((gk + 2g + k + 1)(h + j) + h - z)^2 \\
&\quad - (2n + p + q + z - e)^2 - (16(k+1)^3(k+2)(n+1)^2 + 1 - f^2)^2 \\
&\quad - (e^3(e+2)(a+1)^2 + 1 - o^2)^2 - ((a^2 - 1)y^2 + 1 - x^2)^2 \\
&\quad - (16r^2y^4(a^2 - 1) + 1 - u^2)^2
\end{aligned}$$

3 On remarquera l'analogie avec la machine de Turing universelle $\mathcal{M}_u$.

$$
\begin{aligned}
&-(((a+u^2(u^2-a))^2-1)(n+4dy)^2+1-(x+cu)^2)^2\\
&-(n+1+v+y)^2\\
&-((a^2-1)l^2+1-m^2)^2-(ai+k+1-l-i)^2\\
&-(p+l(a-n-1)+b(2an+2a-n^2-2n-2)-m)^2\\
&-(q+y(a-p-1)+s(2ap-p^2-2p-2)-x)^2\\
&-(z+pl(a-p)+t(2ap-p^2-1)-pm)^2)
\end{aligned}
$$

Toutes les questions encore ouvertes sur les nombres premiers peuvent donc être ramenées à des questions sur les valeurs positives entières du polynôme Q !

On peut également construire de semblables polynômes pour la conjecture de Goldbach[4] ou le grand « théorème » de Fermat.

10.5 Caractère aléatoire d'un programme

Il est bon de citer les résultats de Gregory Chaitin (Chaitin 1979; 1988; 1987a; 1987b) qui éclairent d'un jour différent les problèmes de calculabilité et d'incomplétude des systèmes formels[5].

10.5.1 Hasard et calculabilité

Le but original des travaux qu'a repris Chaitin était d'étudier la notion de hasard, en particulier dans les séries de nombres. Si l'on considère deux séries de nombres, par exemple :

1	3	5	7	9	11	13	15	17	19
7	2	9	1	19	11	14	3	5	6

La première série de nombre n'est visiblement pas aléatoire, alors que la seconde a été obtenue en tirant au hasard des boules numérotées de 1 à 19 dans un sac et nous paraît clairement aléatoire. Cependant, rien ne nous dit que la première série n'a pas été obtenue de la même façon, car, après tout, les probabilités d'obtention des deux séries sont égales : ce n'est donc pas la provenance qui donne le caractère aléatoire d'une série de nombres. En fait, ce que nous exprimons quand nous disons qu'une série est aléatoire est plutôt lié à l'*incompressibilité* de l'information[6] nécessaire à exprimer la série. On peut aisément formaliser cette idée en codant la série de nombres sous forme binaire et en regardant la taille d'un programme chargé de l'exprimer (taille qui s'exprime également en bits). Dans le premier cas le programme serait simplement :

---

4 Tout nombre pair est somme de deux nombres premiers.

5 Ces travaux sont étroitement reliés à ceux de A. Kolmogorov (Kolmogorov 1965; 1968; Li and Vitányi 1990) ou ceux de P. Martin-Lof (Martin-Lof 1966).

6 Cette approche du caractère aléatoire d'une série vient d'ailleurs de la théorie de l'information. Bien entendu, on ne peut négliger les approches classiques pour estimer le caractère aléatoire des séquences de nombres. Voir (Knuth 1981).

```
FOR i:=1 TO 10 DO WRITELN(2*i-1);
```

Dans le second cas, nous n'aurions d'autre ressource que de faire écrire au programme l'ensemble des chiffres de la série. La taille en bits du second programme serait donc de l'ordre de la taille en bits de la série qu'il exprime, alors que la taille du premier serait plus petite. Ce caractère serait évidemment renforcé sur des séries plus longues.

La définition de hasard que l'on peut en dégager est la suivante :

Définition 10.5 – Caractère aléatoire d'une série – *Une suite de nombres est aléatoire si le plus petit algorithme capable de la construire est d'une taille équivalente à la taille de la suite.*

L'existence d'un plus petit algorithme semble évidente. En effet, il existe, pour exprimer un nombre N, une infinité de programmes de taille positive et l'on sait que tout ensemble d'entiers naturels admet une borne inférieure. Remarquons que ce *programme minimal* P est lui-même composé d'une séquence aléatoire de bits, car s'il était généré par un autre programme P' de taille plus petite, alors l'exécution de P' générerait P sur lequel il suffirait de continuer l'exécution pour trouver N avec comme seule information initiale P'. Chaitin appelle le nombre de bits du programme minimal capable d'engendrer une série de bits la *complexité* de la série de bits :

Définition 10.6 – Complexité d'une série – *On appelle complexité d'une série la taille en bits du programme minimal qui peut la générer.*

La question que l'on peut maintenant se poser est la suivante : comment démontrer qu'une suite de nombres est aléatoire ? Il est relativement facile de prouver qu'elle ne l'est pas : il suffit de construire un algorithme permettant de l'exprimer plus efficacement. Mais comment trouver effectivement une suite aléatoire ? La seule chose que nous puissions faire est d'écrire un programme T qui serait : « Trouver une série de bits telle que l'on puisse prouver que sa complexité est plus grande que celle de ce programme T[7] ». Il suffit alors de faire s'exécuter le programme T. Il ne peut pas s'arrêter. En effet, s'il trouve une série de complexité supérieure au programme T, alors cela veut dire que le programme T peut la calculer (c'est ce qu'il vient de faire) et donc que la complexité de la série est exactement celle du programme T : le problème n'est pas calculable.

Ce résultat signifie que, dans un système, on ne peut exprimer qu'une série de bits possède une complexité plus grande que le nombre de bits du programme qui l'exprime. Or, on peut regarder un programme et un calculateur comme un système formel dont les axiomes et les règles d'inférence constituent les règles de calcul ; on en déduit le théorème suivant :

Théorème 10.10 – Théorème de Chaitin – *Soit n la complexité (taille en bits) des axiomes et règles d'inférence d'un système formel. Si la complexité du programme T recherchant une série aléatoire est c, on ne peut prouver qu'une série de bits a une complexité supérieure à $n + c$.*

Comme d'autre part, la complexité est définie en terme de hasard, ce théorème prouve que dans un système formel on ne peut prouver qu'un nombre est aléatoire si sa complexité n'est pas inférieure à celle du système formel lui-même. La complexité du système

---

7 Souvenez-vous de la démonstration du théorème de Gödel : on construit une formule qui exprime sa propre indémontrabilité : ici on construit un programme. Le mécanisme dérive finalement du paradoxe de Berry (le plus petit nombre que l'on peut définir en moins de quinze mots).

est importante car elle représente la quantité d'information que contient le système et donc la quantité d'information qu'il pourra générer. Les axiomes et règles d'inférence sont aléatoires (ils sont minimaux) et sont les règles qui permettent de tester le caractère aléatoire d'autres nombres (qui codent les formules). Le théorème de Chaitin exprime donc qu'il y a dans tout système formel des nombres (formules) indémontrables car de complexité supérieure à celle du système lui-même. Il donne du théorème de Gödel et des problèmes de calculabilité une approche intuitive liée à la notion d'information contenue dans le système formel.

10.5.2 Probabilité d'arrêt d'un programme

Le but de Chaitin était de construire une série de bits qui soit aléatoire. Il définit alors un nombre Ω qui est la probabilité d'arrêt d'un programme aléatoire pour un ordinateur donné. Le programme est une suite finie de 0 et de 1 qui ont été tirés à pile ou face. Le nombre Ω possède la particularité intéressante d'être aléatoire *i.e.*, de ne pouvoir être exprimé par une séquence de bits plus courte que lui-même.

Il est possible de construire des approximations d'Ω en calculant des nombres Ω_n qui représentent la probabilité pour qu'un programme aléatoire de n bits au plus s'arrête en au plus n secondes. Il est alors possible (en théorie) de calculer Ω_n en examinant extensivement combien de programmes n_k de longueur k ($1 \leq k \leq n$) s'arrêtent en n secondes au plus. On a alors :

$$\Omega_n = \sum_{1 \leq k \leq n} 2^{-k} n_k$$

Quand n devient très grand Ω_n tend vers Ω.

Chose amusante, Chaitin, en utilisant les résultats (Jones and Matijasevich 1984) de J. Jones (Université de Calgary) et de Y. Matijasevic (Institut de mathématiques de Léningrad), a construit une famille d'équations diophantiennes qui possèdent la propriété suivante : la k^e équation diophantienne de la série admet une infinité de solutions si le k^e bit de Ω vaut 1, et n'a qu'un nombre fini de solutions sinon.

10.5.3 Complexité organisée

Le physicien américain Charles Bennett s'est inspiré des travaux de Chaitin pour définir, à coté de la complexité définie par Chaitin qu'il appelle *complexité aléatoire*, une complexité organisée. La complexité organisée d'un objet est le temps nécessaire au programme minimal (au sens du Chaitin) pour calculer cet objet.

Ainsi, alors que la suite des 10^{10} premiers nombres premiers a une complexité aléatoire faible (le programme minimal pour les calculer est petit), sa complexité organisée est forte car le temps de calcul est long.

CHAPITRE 11

La complexité

Jean-Marc Alliot — Thomas Schiex

11.1 Introduction

La théorie de la complexité a pour but de donner un contenu formel à la notion intuitive de difficulté de résolution d'un problème. La théorie de la calculabilité nous a permis d'identifier un certain nombre de problèmes non calculables. Nous sommes, d'autre part, aptes à fournir des algorithmes pour d'autres problèmes. Mais le fait que nous puissions les résoudre théoriquement ne nous permet pas d'affirmer que nous saurons effectuer le calcul en pratique, si, par exemple, le temps d'exécution se révèle prohibitif. La théorie de la complexité permet de dégager plusieurs niveaux de difficulté. C'est une aide très appréciable pour déterminer s'il est raisonnable ou non de se lancer dans la résolution de ce problème, ou s'il faut au contraire chercher à le reformuler pour aboutir à un problème plus simple. Il s'agit d'une théorie encore très ouverte aujourd'hui. Nombre de problèmes, dont certains importants, n'ont pas encore trouvé de réponse.

Avant que le lecteur ne consulte les paragraphes suivants[1], nous lui conseillons de relire rapidement le chapitre consacré aux machines de Turing et spécialement les quelques lignes traitant des machines de Turing non déterministes.

11.2 Définitions fondamentales

11.2.1 Problèmes de décision et langages

De façon informelle, un problème Π est défini par l'ensemble des propriétés que doivent vérifier ses *solutions*. Généralement, ces propriétés font apparaître un ou plusieurs paramètres non spécifiés. Une instance I d'un problème Π est obtenue en spécifiant des valeurs particulières pour tous les paramètres du problème.

---

1 Voici quelques ouvrages traitant de la théorie de la complexité : la référence la plus classique est sans aucun doute (Garey and Johnson 1979). (Salomaa 1989) est assez éclectique, il traite également les problèmes de théorie des langages, de calculabilité, de cryptographie, de réseaux de Pétri. (Papadimitriou 1994) suit une approche similaire, avec des résultats plus récents. Dans (Johnson 1990), le lecteur trouvera un résumé synthétique mais un peu sec des principaux résultats et classes définies. En français, on pourra consulter (Wolper 1991) ou (Barthélemy *et al.* 1992). Ce dernier présente une approche intéressante de la notion de codage raisonnable.

Définition 11.1 – *On dira qu'un algorithme résout un problème* Π *s'il peut être appliqué à toute instance* I *de* Π *pour fournir la solution de l'instance* I *en un temps fini.*

Les problèmes qui nous intéresseront sont, pour des raisons de facilité, des problèmes de décisions *i.e.*, des problèmes qui n'ont que deux solutions possibles : la réponse *oui* ou la réponse *non*. Si l'on note D_Π l'ensemble des instances possibles d'un problème *de décision* Π, on peut distinguer le sous-ensemble Y_Π (pour *yes*) des instances pour lesquelles la réponse est *oui* du sous-ensemble N_Π des instances pour lesquelles la réponse est *non*.

Exemple de problème de décision : « un entier positif k étant donné, k est il premier ? »

On définit simplement le complémentaire d'un problème (de décision) :

Définition 11.2 – Complémentaire – *Étant donné un problème de décision* Π, *on définit son complémentaire (noté* Π^c*) par :*

$$D_{\Pi^c} = D_\Pi, Y_{\Pi^c} = N_\Pi, N_{\Pi^c} = Y_\Pi$$

i.e., *le problème complémentaire* Π^c *d'un problème de décision* Π *est défini par la question opposée de celle définissant* Π.

Le complémentaire du problème précédent s'énonce donc : « un entier positif k étant donné, existe-t-il deux entiers $m, n > 1$ tels que $k = m.n$? ».

Se restreindre aux problèmes de décision n'est pas si limitatif qu'il pourrait paraître au premier abord : un problème de recherche pourra se transformer en problème d'existence, un problème d'optimisation en un problème d'existence d'une solution de qualité suffisante. En fait, la raison de cette limitation aux problèmes de décision est qu'il existe une contrepartie naturelle, mais formelle, à un problème de décision, il s'agit du problème de reconnaissance d'un mot dans un langage.

La correspondance entre problèmes de décisions et langages s'effectue au moyen d'une fonction de codage e, permettant de traduire toute instance d'un problème de décision Π en un mot sur un alphabet Σ. L'ensemble des mots de Σ^* est alors partitionné par e et Π en :

- l'ensemble des mots qui ne sont pas des codages d'une instance de Π, qui ne nous intéressent pas ;
- l'ensemble des mots qui résultent du codage d'une instance de N_Π, pour laquelle la réponse est *non* ;
- l'ensemble des mots, noté $L[\Pi, e]$, qui résultent du codage d'une instance de Y_Π, pour laquelle la réponse est *oui*.

La notion informelle d'algorithme permettant de résoudre un problème de décision peut ensuite être ramenée à la notion formelle de programme de machine de Turing (voir chapitre 4) acceptant un mot $x \in \Sigma^*$, x résultant du codage d'une instance I. Plus précisément, nous considérerons des machines de Turing utilisant un alphabet Γ contenant Σ et possédant *deux* états d'arrêt z_Y et z_N. Appliquée à un mot x, une telle machine $\mathcal{M}$ peut :

- s'arrêter dans l'état z_Y, on dira alors que le mot est accepté par la machine. L'ensemble des mots acceptés est noté $L(\mathcal{M})$;
- s'arrêter dans l'état z_N, on dira alors que le mot est rejeté par la machine ;
- ne pas s'arrêter.

Cependant, et pour que la notion de programme de machine de Turing corresponde précisément à la notion d'algorithme qui résout un problème, nous ne considérerons par la suite que des programmes qui s'arrêtent sur toute entrée $x \in \Sigma^*$. Nous ne nous intéresserons donc pas, dans ce chapitre, à des problèmes indécidables.

Définition 11.3 – *On dira qu'une machine de Turing $\mathcal{M}$ résout le problème de décision Π, sous le codage e, si $\mathcal{M}$ s'arrête sur toute entrée $x \in \Sigma^*$ et si l'ensemble des mots acceptés par $\mathcal{M}$ est égal à l'ensemble $L[\Pi, e]$.*

Définition 11.4 – Efficacité d'un algorithme – *Établir l'efficacité d'un algorithme, c'est donner une relation entre le nombre d'opérations effectuées par une machine de Turing $\mathcal{M}$ exécutant l'algorithme A et la taille m de l'argument de la machine de Turing.*

Le problème maintenant est de définir la notion de taille des arguments. En effet, cette taille est dépendante de la fonction de codage et en particulier de l'alphabet utilisé pour le codage d'un argument en un mot. Ainsi le nombre 10 codé en notation décimale à une longueur de 2 caractères, de 4 caractères en numération binaire et de 10 caractères en notation unaire. Toutes les propriétés que nous énoncerons par la suite supposent qu'un codage « raisonnable » est employé. Informellement, il s'agit d'un codage concis, sans redondance, et décodable. Dans le cas des nombres, on peut en donner une définition plus précise[2] :

Définition 11.5 – Codage raisonnable des données – *Le codage des données est raisonnable si les nombres ne sont pas codés sous forme unaire. L'encombrement mémoire doit être de la forme* $\log n$. *Nous noterons la taille d'un mot codé (sa longueur)* $|n|$.

Ainsi, au sens de la complexité, il est indifférent qu'un nombre soit codé en base 2,3,10 ou 16. En revanche, il *ne doit pas* être codé sous forme unaire. Nous verrons qu'il y a d'excellentes raisons à cela.

Définition 11.6 – Complexité temporelle pour une machine déterministe – *La complexité temporelle d'un programme de machine de Turing déterministe $\mathcal{M}$ est la fonction de n :*

$$T_{\mathcal{M}}(n) = \max_{x \in \Sigma^*, |x|=n} \{k \mid k \textit{ longueur de calcul sur l'entrée } x\}$$

La notion de complexité temporelle est intuitivement pertinente : nous considérons l'ensemble des mots de longueur n et nous regardons quel est celui qui entraîne le calcul le plus long. Nous sommes alors sûrs que tous les autres mots de longueur n seront traités dans un temps inférieur, par définition.

Définition 11.7 – Complexité temporelle pour une machine non déterministe – *La complexité temporelle d'un programme d'une machine de Turing non déterministe $\mathcal{M}$ est la fonction de n :*

$$T_{\mathcal{M}}(n) = \max_{x \in L(\mathcal{M}), |x|=n} \{\min\{k \mid k \textit{ longueur d'un calcul d'acceptation de } x\}\}$$

---

2 Une définition plus large, s'appliquant en sus aux graphes, fonctions et ensembles finis est proposée dans (Garey and Johnson 1979). La notion de codage *sympathique*, plus précise, est présentée dans (Barthélemy *et al.* 1992).

La définition de la complexité temporelle d'un programme d'une machine de Turing non déterministe est plus délicate. On prend le maximum sur l'ensemble des mots de taille n, mais pour chaque mot, on considère que l'on suit le chemin le plus court menant à une acceptation dans l'arbre de résolution de la machine de Turing non déterministe. Il faut noter l'asymétrie, introduite par le non déterminisme, entre acceptation et rejet, le maximum étant calculé sur les mots *acceptés* seulement. Nous y reviendrons.

11.3 Problèmes faciles et intraitables

Dans cette section, nous allons nous intéresser à une première classe de problèmes, celle des problèmes polynomiaux[3].

11.3.1 Problèmes polynomiaux

Définition 11.8 – Problèmes polynomiaux – *L'ensemble des problèmes polynomiaux (ou l'ensemble des langages reconnus en un temps polynomial par une machine de Turing déterministe) est l'ensemble*

$$\mathsf{P} = \{L | L \subset \Sigma^*, \exists \mathcal{M}, L = L(\mathcal{M}), \exists P, \forall x \in L, |x| = n, T_{\mathcal{M}}(x) < P(n)\}$$

où P est une fonction polynomiale et M une machine de Turing déterministe.

Cette classe de problèmes (ou langages) est appelée classe P.

L'ensemble des problèmes[4] polynomiaux est donc l'ensemble des problèmes dont le temps de résolution est borné par un polynôme fonction de la taille des données. On comprend mieux maintenant pourquoi on exige un codage raisonnable, et en particulier, de ne pas utiliser de représentation unaire pour les nombres. Considérons en effet une donnée numérique N pour un problème polynomial codée dans le système décimal, en base 2 et en base 1. Dans le premier cas, la longueur du codage sera de l'ordre de $\log_{10}(N)$, dans le second, de l'ordre de $\log_2(N)$ alors que dans le dernier cas elle sera de l'ordre de N lui-même, qui ne peut pas s'exprimer sous la forme d'un polynôme fonction de $\log_2(N)$ ou de $\log_{10}(N)$. Ainsi, un problème de complexité polynomiale en utilisant un codage en base 1 n'est pas nécessairement polynomial pour des codages en base 2 ou plus, alors que tout problème polynomial en utilisant un codage en base $i \geq 2$ reste polynomial si un codage en une autre base $j \geq 2$ est employé : *l'ensemble des problèmes polynomiaux est invariant par changement de fonction de codage, pourvu que le codage soit un codage raisonnable.*

Il existe une sous-classe des problèmes polynomiaux, les problèmes linéaires (P est une fonction linéaire), qui constitue une classe de problèmes plus simples que ceux de P. Appartiennent à cette classe des problèmes triviaux comme le calcul de la somme de deux nombres de p chiffres...

---

3 appelés aussi problèmes faciles par certains auteurs.

4 Nous employons indifféremment dans ce chapitre la notion de *langage accepté par une machine de Turing* et la notion de *problème résolu par une machine de Turing* de façon à bien rappeler au lecteur l'interchangeabilité des deux notions. À un problème Π donné, on associe simplement le langage $L[\Pi, e]$ où e est un codage raisonnable.

On définit également co-P, formé de l'ensemble des problèmes (ou langages) complémentaires de problèmes de P. Une machine de Turing déterministe qui sait dire, en un temps polynomial, si un mot représente, ou non, une instance de Y_Π, permet directement de résoudre le problème Π^c en un temps polynomial. Il suffit d'échanger les deux états z_Y et z_N dans son programme. On en déduit donc que :

Théorème 11.1 – P = *co*-P

La classe polynomiale comprend quelques problèmes classiques dont voici des exemples :

- tri d'un ensemble de n nombres ;
- test de planarité d'un graphe[5] ;
- recherche des composantes connexes d'un graphe[6] ;
- recherche de la plus courte chaîne (ou chemin si le graphe est dirigé) simple permettant de se rendre d'un sommet P à un sommet Q dans un graphe dont les arêtes sont valuées positivement[7] ;
- recherche d'une chaîne qui passe par toutes les arêtes d'un graphe[8].

La classe des problèmes polynomiaux est à la fois importante et limitée. Importante parce que nombre de problèmes pratiques indispensables au bon fonctionnement de l'informatique ont des solutions polynomiales, limitée parce que pour nombre de problèmes intéressants, personne n'a réussi à montrer leur caractère polynomial.

11.3.2 Les problèmes prouvés intraitables

L'ensemble des problèmes dont on a prouvé qu'ils étaient calculables mais non polynomiaux est appelé *classe des problèmes prouvés intraitables*[9].

Un problème prouvé intraitable est la théorie formelle de l'addition (théorème de Presburger) : la théorie formelle de l'addition est un sous-système formel de $\mathbb{N}$ comprenant un sous-ensemble du calcul des prédicats et les axiomes de l'addition. Alors que $\mathbb{N}$ (et le calcul des prédicats) est indécidable, ce système est lui décidable, mais intraitable.

11.3.3 Conclusion

Nous avons dégagé plusieurs classes de problèmes, dont les problèmes polynomiaux et les problèmes prouvés intraitables. Mais, il reste tout un ensemble de problèmes que nous ne pouvons pas classer dans P, car on ne connaît pas d'algorithme polynomial permettant de les résoudre, et que nous ne pouvons pas non plus considérer comme intraitables, car on n'a pas prouvé qu'il n'existe pas un algorithme polynomial permettant

---

5 Un graphe est dit planaire si on peut le dessiner sur une feuille sans que ses arêtes se rencontrent en dehors des sommets.

6 Ensemble de points tel qu'il existe toujours une chaîne permettant de passer d'un point de la composante à un autre. On parle aussi d'ensemble de points reliés entre eux (pas forcément deux à deux).

7 Le problème n'est plus polynomial s'il y a des arêtes valuées négativement. Les algorithmes polynomiaux de calcul d'un plus court chemin ne fournissent un chemin *simple* que si le graphe ne possède pas de circuits absorbants (circuits de longueur négative).

8 Problème dit du cycle eulérien, car Euler fut le premier à le résoudre. Une instance fameuse de ce problème est le « problème des ponts de Kœnigsberg » (Berge 1970).

9 *Intractable* en anglais.

de les résoudre. L'introduction de la classe NP et de la notion de problèmes NP-complets permet de pallier, en partie, l'absence de ces résultats.

11.4 Résultats généraux sur les problèmes NP

11.4.1 Problèmes NP

Définition 11.9 – Problèmes NP – *On définit l'ensemble des problèmes non-déterministes polynomiaux (ou ensemble des langages reconnus en un temps polynomial par une machine de Turing non déterministe) par :*

$$\mathsf{NP} = \{L | L \subset \Sigma^*, \exists \mathcal{M}, L = L(\mathcal{M}), \exists P, \forall x \in L, |x| = n, T_{\mathcal{M}}(x) < P(n)\}$$

où P est une fonction polynomiale et $\mathcal{M}$ une machine de Turing non-déterministe.

Cette classe de problèmes est appelée classe NP.

La différence avec la définition de la classe P est que la machine $\mathcal{M}$ est *non-déterministe*. Les problèmes de NP corespondent bien à la notion de casse-tête : il est possible de construire une « solution » en un temps polynomial, mais tout le problème est de trouver cette « solution » (ou plus précisément, un calcul d'acceptation, qui montre qu'il y a une solution). On caractérise donc les problèmes de NP par cette propriété : ce sont les problèmes de décision dont on peut facilement (en un temps polynomial) vérifier, sur une machine déterministe, qu'une solution potentielle est effectivement une solution *i.e.*, qu'un calcul d'acceptation est bien un calcul d'acceptation.

Une machine déterministe étant un cas particulier de machine non déterministe, il ressort de la définition des deux classes P et NP que $\mathsf{P} \subseteq \mathsf{NP}$. *On ne sait pas aujourd'hui si l'inclusion est stricte ou non.* Il s'agit probablement là d'un des problèmes les plus importants de l'informatique théorique moderne, car la classe des problèmes NP est très importante.

On définit, comme pour P, la classe co-NP formée des complémentaires des problèmes (ou langages) de NP. Cependant, la définition de la complexité temporelle d'une machine non-déterministe fait apparaître une asymétrie entre les instances sur lesquelles on répond *oui*, et les autres : on sait que pour les instances pour lesquelles on répond *oui*, il existe un calcul de longueur polynomiale qui permet de répondre *oui* (d'accepter le codage de l'instance), mais on ne sait rien sur les autres. À l'heure actuelle, on ne sait donc pas si $\mathsf{NP} = \text{co-NP}$. En fait, la conjecture $\mathsf{NP} \neq \text{co-NP}$ est plus forte encore que la conjecture $\mathsf{NP} \neq \mathsf{P}$, car du fait $\mathsf{P} = \text{co-P}$, $\mathsf{NP} \neq \text{co-NP}$ impliquerait $\mathsf{NP} \neq \mathsf{P}$ (en effet, si $\mathsf{P} = \mathsf{NP}$, alors $\mathsf{P} = \text{co-P} = \text{co-NP} = \mathsf{NP}$).

11.4.2 Problèmes NP-complets

Définition 11.10 – Transformation polynomiale – *Une* transformation polynomiale[10] *d'un language $L_1 \subset \Sigma_1^*$ en un language $L_2 \subset \Sigma_2^*$ est une fonction $f : \Sigma_1^* \to \Sigma_2^*$ qui satisfait les deux conditions suivantes :*

10 Aussi appelée *many-one reduction* dans certaines références en langue anglaise.

1. *f peut être calculée en un temps polynomial par une machine de Turing déterministe ;*
2. *pour tout $x \in \Sigma_1^*, x \in L_1$ si et seulement si $f(x) \in L_2$.*

Définition 11.11 – Réductibilité – *Un langage $L_1 \subset \Sigma_1^*$ est polynomialement réductible à un langage $L_2 \subset \Sigma_2^*$ ssi il existe une transformation polynomiale de L_1 en L_2. On note alors $L_1 \propto_p L_2$.*

Notons que $\propto_p$ est un préordre (une relation réflexive et transitive). Nous dirons que deux langages L_1 et L_2 sont *polynomialement équivalents* quand $L_1 \propto_p L_2$ et $L_2 \propto_p L_1$. La relation ainsi définie est une relation d'équivalence dont les classes d'équivalence sont ordonnées par $\propto_p$. Les problèmes NP-complets forment alors la classe d'équivalence des problèmes les plus difficiles de NP :

Définition 11.12 – Problème NP-complet – *Soit un langage L. L est dit NP-complet ssi*
- $L \in \mathsf{NP}$
- $\forall L' \in \mathsf{NP}, L' \propto_p L$

Les langages NP-complets[11] représentent donc les problèmes NP les plus difficiles à résoudre puisque tout problème de NP peut se ramener à un problème NP-complet en un temps polynomial. Nous avons d'autre part le théorème suivant :

Théorème 11.2 – *Soit L_0 un langage NP-complet alors :*

$$L_0 \in \mathsf{P} \Leftrightarrow \mathsf{P} = \mathsf{NP}$$

Il est clair que si $\mathsf{P} = \mathsf{NP}$ alors L_0 est dans P.

Réciproquement, soit L_0 NP-complet et $L_0 \in \mathsf{P}$. Nous savons qu'il existe une machine de Turing déterministe $\mathcal{M}$ qui accepte L_0 en un temps borné par un polynôme P. Pour tout langage de L de NP, nous savons qu'il existe f telle que $f(L) = L_0$ et f calculable en un temps borné par un polynôme Q. Soit $x \in L$, nous pouvons alors :

1. calculer $f(x)$ en un temps $Q(|x|)$;
2. ramener la tête de lecture sur le premier symbole en un temps $|f(x)|$;
3. décider si $f(x) \in L_0$ en appliquant $\mathcal{M}$ en un temps $P(|f(x)|)$.

Le temps total de calcul T est

$$T \leq Q(|x|) + |f(x)| + P(|f(x)|)$$

mais nous savons d'autre part que : $|f(x)| \leq |x| + Q(|x|)$ car la machine de Turing en $Q(|x|)$ opérations n'a pas pu écrire plus de $Q(|x|)$ symboles. On a :

$$T \leq 2Q(|x|) + |x| + P(Q(|x|) + |x|)$$

Donc il existe un polynôme R tel que :

$$T \leq R(|x|)$$

---

11 La catégorie de problèmes qui ne vérifient pas la première partie de la définition (qui ne sont pas forcément dans NP) sont dits (ce ne sont pas les seuls) NP-difficiles, ou NP-durs.

Donc tout problème de NP est soluble en un temps polynomial, donc NP = P.

Ce théorème est un théorème capital en théorie de la complexité. Il signifie en effet que, si nous sommes capables de trouver un algorithme polynomial pour un problème NP-complet, alors tous les problèmes NP-complets seront solubles en un temps polynomial. De nombreux mathématiciens et informaticiens travaillent sur le sujet depuis une vingtaine d'années maintenant, et il semble probable, bien que non démontré, que NP $\neq$ P.

Théorème 11.3 – *Soit* L_0 NP*-complet. Si* $L_0 \propto_p L_1$ *et* $L_1 \in$ NP, L_1 *est* NP*-complet.*

Il nous suffit de démontrer que pour tout $L \in$ NP, $L \propto_p L_1$. Nous savons que L_0 est NP-complet, donc pour tout L de NP, il existe une transformation polynomiale f telle que $f(L) = L_0$. D'autre part, puisque nous avons $L_0 \propto_p L_1$, il existe une transformation polynomiale g telle que $g(L_0) = L_1$. Donc la transformation $h = g \circ f$ est polynomiale et vérifie :

$$h(L) = L_1$$

Q.E.D

Ce théorème est également fondamental, car une fois connu un problème L_0, NP-complet, il suffit pour démontrer qu'un problème L est NP-complet de montrer qu'il est dans NP[12] et de trouver une transformation polynomiale réduisant L_0 à L. En fait, il s'agit là du même type de démonstration que nous avons utilisé en théorie de la calculabilité : nous ramenons toujours un problème nouveau à un problème connu : les démonstrations directes sont rarissimes.

Nous avons donc dégagé une nouvelle classe de problèmes et nous commençons même à avoir des outils pour les étudier. Il nous reste encore une chose bien difficile à faire : trouver un problème qui appartienne effectivement à la classe des problèmes NP-complets. Ce problème ne fut résolu qu'en 1971. C'est l'objet du théorème de Cook.

11.4.3 Le théorème de Cook

Nous rappelons brièvement la définition de la satisfiabilité d'une formule sous forme conjonctive normale.

Définition 11.13 – Satisfiabilité d'une formule – *Soit une formule F du calcul propositionnel sous forme conjonctive normale utilisant les variables propositionnelles* $(x_1 \ldots x_n)$.

Résoudre le problème de la satisfiabilité de F consiste à déterminer s'il existe une interprétation des x_i telle que F soit satisfaite (F vraie). Ce problème est appelé problème SAT (ou langage SAT)[13].

Toutes les notions apparaissant dans cette définition ont été introduites en logique, nous ne revenons pas dessus.

Théorème 11.4 – Théorème de Cook – *Le langage SAT est* NP*-complet.*

---

12 Cette propriété est souvent simple à établir. Il existe cependant quelques cas difficiles : le problème de satisfiabilité dans le système modal propositionnel S_5 par exemple (cf. (Ladner 1977)), qui est NP-complet.

13 Certains auteurs réservent le terme de SAT au problème de satisfiabilité d'une formule quelconque (habituellement appelé GENERAL SAT). Le problème de satisfiabilité d'un ensemble de clauses est alors appelé CONSAT.

La démonstration de ce théorème est assez longue, mais aussi très instructive. Elle ne présente aucune difficulté théorique majeure. Le lecteur pressé ou dégoûté par les mathématiques peut s'en dispenser sans préjudice.

Il nous suffit de prouver que pour tout langage L de NP, on a $L \propto_p \text{SAT}$[14].

Le principe de la méthode est simple. Pour tout mot m, nous allons construire un algorithme polynomial transformant m en une formule F qui sera satisfiable ssi il existe un calcul d'acceptation de m en un temps polynomial $P(n)$ (avec $n = |m|$) par une machine de Turing $\mathcal{M}$ non déterministe. Nous aurons ainsi prouvé $L \propto_p \text{SAT}$.

Nous noterons $i, 0 \leq i \leq p$ les états de la machine de Turing (0 sera l'état de démarrage et p correspondra à l'état d'acceptation z_Y), et $s_j, 1 \leq j \leq q$ les symboles de l'alphabet Σ (le symbole B sera noté B). Les cases de la bande seront désignées par la lettre c et seront numérotées par des entiers relatifs.

Soit un mot $m = (a_1 a_2 \ldots a_n) \in \Sigma^*$. Posons $|m| = n$. Le mot m est dans $L \in \text{NP}$ ssi il existe une séquence de calcul de $\mathcal{M}$ qui aboutit à un succès en un temps $P(n)$[15]. Nous voyons donc que nous n'aurons à considérer que des instants t tels que :

$$0 \leq t \leq P(n)$$

D'autre part, la machine de Turing étant dans une configuration valide au départ, la tête de la machine de Turing pointe sur la case 0. Comme elle ne pourra pas effectuer plus de $P(n)$ déplacements en $P(n)$ étapes, nous n'avons à considérer que les cases c telles que :

$$-P(n) \leq c \leq P(n)$$

Les deux conditions que nous venons d'énoncer limitent l'espace de notre problème. Il nous reste à énoncer les conditions liées au fonctionnement de la machine de Turing non déterministe et à les traduire en formule logique.

Nous allons utiliser comme atomes de notre formule l'ensemble de variables logiques :

$$[c, t, s, z]$$

Cette variable prend la valeur vraie si la case c est à l'instant t dans l'état (s, z). On définit l'état d'une case c à l'instant t comme étant :

(s,-1) si c contient le symbole s à t et que la machine de Turing pointe sur une autre case que c.

(s,z) si c contient le symbole s à l'instant t et que la machine de Turing pointe sur la case c et est dans l'état z, $0 \leq z \leq p$.

Le nombre de variables est égal au nombre de cases $(2P(n) + 1)$, multiplié par le nombre d'instants possibles $(P(n)+1)$, multiplié par le nombre de lettres de l'alphabet (q) et le nombre d'états de la machine de Turing plus un, soit $(p + 2)$. Le nombre de variables est donc de l'ordre[16] de $(P(n))^2$.

Nous sommes maintenant en mesure de traduire la suite des conditions qui exprime qu'un mot $m \in L$ ssi il existe une séquence de calcul de la machine de Turing non-déterministe $\mathcal{M}$ qui reconnaît m.

---

14 Il faut aussi démontrer que SAT est dans NP : il suffit de considérer une machine de Turing non-déterministe qui à chaque étape donne à une variable non encore affectée la valeur 0 ou la valeur 1 et évalue la formule lorsque toutes les variables ont été affectées.

15 En toute rigueur, elle aboutit en un temps inférieur ou égal à $P(n)$. Mais on peut toujours supposer que le calcul ne se termine qu'à l'instant $P(n)$ en imposant à la machine de Turing de ne rien faire dès qu'elle a terminé le calcul effectif, jusqu'à $P(n)$, instant de l'arrêt.

16 Nous ne nous intéressons qu'à la complexité en n.

1. À l'instant initial, $\mathcal{M}$ est dans la configuration initiale définie par m. Donc les cases $0, 1, \ldots, n-1$ sont occupées par les symboles $a_1, a_2, \ldots, a_n$ et les cases $-P(n), -P(n+1), \ldots, -1$ et $n, n+1, \ldots, P(n)$ sont occupées par le symbole blanc B. D'autre part la machine est dans l'état initial z_0, noté 0. On peut traduire cet état par la formule :

$$F_1 = (\bigwedge_{i=-P(n)}^{-1} [i, 0, B, -1]) \wedge [0, 0, a_1, 0] \wedge (\bigwedge_{i=1}^{n-1} [i, 0, a_{i+1}, -1]) \wedge (\bigwedge_{i=n}^{P(n)} [i, 0, B, -1])$$

Cette formule peut être écrite en un nombre de pas de l'ordre de $P(n)$[17].

2. À l'instant $P(n)$, la machine de Turing a terminé son calcul. Elle est alors dans l'état final z_Y (noté p) et pointe sur une case quelconque contenant un symbole quelconque. On peut donc le traduire par :

$$F_2 = \bigvee_{i=-P(n)}^{P(n)} \bigvee_{j=1}^{q} [i, P(n), s_j, p]$$

Cette formule peut être écrite en un nombre de pas de l'ordre de $P(n)$[18] (q est une constante).

Nous avons avec les conditions (1) et (2) traduit l'existence d'un état initial défini par m et d'un état final d'acceptation. Nous allons maintenant exprimer que le changement d'état d'une case dépend de la fonction de transition δ[19] de la machine de Turing $\mathcal{M}$.

3. Considérons une case c dans un état (s, z) avec $0 \leq z \leq p$. Que devient cette case à l'instant $t+1$? Elle peut :
   (a) passer dans l'état (s', z') si $(z', s', I) \in \delta(z, s)$;
   (b) passer dans l'état $(s >>, -1)$ si $(z >>, s >>, G) \in \delta(z, s)$ ou $(z >>, s >>, D) \in \delta(z, s)$.

   Il faut bien se rappeler que la machine de Turing est non déterministe. Donc à chaque étape, elle a plusieurs possibilités d'action (la fonction δ a pour image un ensemble de triplets et non un seul triplet). Ainsi non seulement nous avons deux possibilités principales (I) ou (G ou D), mais pour chacune de ces possibilités, il peut y avoir plusieurs alternatives. Cette condition peut se traduire par la formule :

$$f_3(c, t) = [c, t, s, z] \rightarrow (\bigvee_{i \in \mathcal{D}_1} [c, t+1, s(i), z(i)]) \vee (\bigvee_{j \in \mathcal{D}_2} [c, t+1, s(j), -1])$$

Le domaine $\mathcal{D}_1$ décrit le sous-ensemble de $\delta(z, s)$ comprenant les actions qui laissent la machine immobile, et le domaine $\mathcal{D}_2$ le sous-ensemble des actions qui déplacent la machine à droite ou à gauche. La taille d'un programme donné d'une machine de Turing étant finie, elle est bornée par une constante C, donc la formule f_3 précédente peut être

---

17 En fait, en toute rigueur, la notation des variables étant dyadique, il faut prendre $(\log n)P(n)$ comme majorant ou, pour être tranquille, $nP(n)$. Mais cela ne change en rien le caractère polynomial de l'expression.

18 La note précédente s'applique et continuera de s'appliquer pour les étapes suivantes.

19 Pour le lecteur qui aurait oublié ce qu'est la fonction de transition δ, un petit retour à la définition des machines de Turing non-déterministes est conseillé.

écrite en au plus C pas. D'autre part, nous devons écrire une telle formule pour tous les c et tous les t. La formule F_3 s'écrit donc finalement :

$$F_3 = \bigwedge_{\substack{-P(n)\leq c\leq P(n)\\ 0\leq t\leq P(n)}} f_3(c,t)$$

Nous avons là une opération qui s'effectuera en un nombre de pas de l'ordre de $(P(n))^2$.

4. Considérons maintenant une case c dans l'état $(s,-1)$ à l'instant t. À l'instant $t+1$, cette case peut :
   (a) rester dans l'état $(s,-1)$;
   (b) passer dans l'état (s,z) si la machine de Turing est à l'instant t dans un état z' et pointe sur la case $c+1$ qui contient un symbole s' tel que : $(z,s,G)\in\delta(z',s')$;
   (c) passer dans l'état (s,z) si la machine de Turing est à l'instant t dans un état z' et pointe sur la case $c-1$ qui contient un symbole s' tel que : $(z,s,D)\in\delta(z',s')$.

   Tout cela peut se traduire par la formule :

$$\begin{aligned} f_4(c,t) = {} & [c,t,s,-1]\rightarrow \\ & [c,t+1,s,-1]\vee \\ & \Big(\bigvee_{i\in\mathcal{D}_1}([c,t+1,s,z(i)]\wedge[c+1,t,s'(i),z'(i)])\Big)\vee \\ & \Big(\bigvee_{j\in\mathcal{D}_2}([c,t+1,s,z(j)]\wedge[c+1,t,s'(j),z'(j)])\Big) \end{aligned}$$

Le domaine $\mathcal{D}_1$ recouvre l'ensemble des éléments du graphe de δ tels que $\exists z,z'$, $\exists s,s',(z,s,G)\in\delta(z',s')$ et $\mathcal{D}_2$ recouvre l'ensemble des éléments du graphe de δ tels que $\exists z,z',s,s',(z,s,D)\in\delta(z',s')$. Comme précédemment nous pouvons établir que la formule f_4 peut s'écrire en au plus C pas avec C constant. Nous devons là aussi écrire une telle formule pour tous les c et tous les t. La formule F_4 s'écrit donc finalement :

$$F_4 = \bigwedge_{\substack{-P(n)\leq c\leq P(n)\\ 0\leq t\leq P(n)}} f_4(c,t)$$

Nous avons là une opération qui s'effectuera en un nombre de pas de l'ordre de $(P(n))^2$.

Les deux conditions précédentes sont insuffisantes. Il nous faut aussi exprimer que l'on n'a pas le droit, dans une séquence de calcul, de lire deux cases à la fois, comme le permettent les équations précédentes. Ce sont les conditions que nous allons rédiger maintenant.

5. Nous allons tout d'abord exprimer que pour une case c prise à un instant t, il existe exactement un (s,z) tel que c soit dans l'état (s,z) (z pouvant valoir -1) à l'instant t. Nous voulons donc exprimer un « ou » exclusif sur les variables de la forme $[c,t,s_i,j]$, car nous voulons avoir $[c,t,s_1,-1]\vee[c,t,s_1,0]\vee\ldots\vee[c,t,s_q,p]$ mais nous ne voulons pas avoir simultanément deux $[c,t,s_i,j]$. Rappelons qu'en logique, le ou exclusif sur A,B,C se traduit par $(A\vee B\vee C)\wedge((\neg A\vee\neg B)\wedge(\neg B\vee\neg C)\wedge(\neg A\vee\neg C))$. Cette formule s'étend aisément à un nombre quelconque de termes. Dans notre cas, la formule que nous obtenons est donc :

$$f_5(c,t) = \Big(\bigvee_{\substack{1\leq i\leq q\\ -1\leq j\leq p}}[c,t,s_i,j]\Big)\wedge\Big(\bigwedge_{\substack{1\leq i,k\leq q\\ -1\leq j,l\leq p}}(\neg[c,t,s_i,j]\vee\neg[c,t,s_k,l])\Big)$$

Cette formule a une taille bornée par une constante (p et q sont constants). D'autre part, nous devons écrire une telle équation pour toutes les cases c à tous les instants t. La formule définitive est donc :

$$F_5 = \bigwedge_{\substack{-P(n)\leq c\leq P(n)\\ 0\leq t\leq P(n)}} f_5(c,t)$$

Nous avons là une opération qui s'effectuera en un nombre de pas de l'ordre de $(P(n))^2$.

6. Nous voulons maintenant exprimer que pour un t fixé, il existe une case c contenant un symbole quelconque, telle que c est dans un des états $i, 0 \leq i \leq p$, c'est-à-dire un état valide (différent de -1) de la machine de Turing. Nous pouvons l'exprimer par :

$$f_6(t) = \bigvee_{\substack{-P(n)\leq c\leq P(n)\\ 1\leq i\leq q, 0\leq j\leq p}} [c,t,s_i,j]$$

Cette formule contient un nombre d'éléments de l'ordre de $P(n)$. Nous devons d'autre part écrire cette formule pour tous les instants t. Donc, la formule définitive est :

$$F_6 = \bigwedge_{0\leq t\leq P(n)} f_6(t)$$

Nous avons là une opération qui s'effectuera en un nombre de pas de l'ordre de $(P(n))^2$ en tout.

7. Il nous reste maintenant à exprimer le fait que, pour un t donné, une seule case c peut être dans un état (s,z) avec z différent de -1. Donc si une case est dans un état (s,z), $z \neq -1$ à t toutes les autres sont dans l'état $(s',-1)$ au même instant. Ceci peut s'écrire :

$$f_7(c,t,s,z) = [c,t,s,z] \rightarrow \bigwedge_{\substack{-P(n)\leq c'\leq P(n)\\ c'\neq c}} \bigvee_{1\leq i\leq q} [c',t,s_i,-1]$$

Cette formule contient un nombre d'éléments de l'ordre de $P(n)$. Nous devons d'autre part écrire cette formule pour tous les instants t, toutes les cases c, tous les symboles s et tous les états z différents de -1. Donc, la formule définitive est :

$$F_7 = \bigwedge_{\substack{0\leq t\leq P(n), 0\leq z\leq p\\ -P(n)\leq c\leq P(n), 1\leq s\leq q}} f_7(c,t,s,z)$$

Nous avons là une opération de construction qui s'effectuera en un nombre de pas de l'ordre de $(P(n))^3$ en tout $(P(n)\times(P(n))^2)$.

Nous voici au bout de nos peines. La formule :

$$F = \bigwedge_{i=1}^{7} F_i$$

est satisfiable ssi il existe un calcul d'acceptation de longueur polynomiale du mot m par la machine $\mathcal{M}$. D'autre part, la transformation du problème de reconnaissance de m en la formule F s'effectue en un nombre de pas de l'ordre de $(P(n))^3$ et est donc polynomiale. Donc tout langage $L \in$ NP est réductible au problème consistant à savoir si une formule est satisfiable. Donc SAT est NP-complet.

11.4.4 Exemples de problèmes NP-complets

La classe des problèmes NP-complets est très importante et nous ne citerons que quelques-uns de ses éléments, les plus importants ou les plus intéressants. Nous ne donnons pas toujours les démonstrations, mais nous essayons de les indiquer chaque fois qu'elles ne sont pas trop longues.

Le problème 3-SAT : Il s'agit du problème SAT dans lequel on impose à chacune des clauses de ne comporter au plus qu'une disjonction de trois littéraux.

Il nous suffit, pour démontrer que ce problème est NP-complet, de démontrer que le problème SAT est polynomialement réductible à 3-SAT.

La démonstration est là encore très simple. Il suffit de remarquer que toute clause :

$$x_1 \vee x_2 \vee \ldots \vee x_n$$

peut se mettre sous la forme :

$$(x_1 \vee x_2 \vee y_1) \wedge (x_3 \vee \neg y_1 \vee y_2) \wedge \ldots \wedge (x_{n-2} \vee \neg y_{n-4} \vee y_{n-3}) \wedge (x_{n-1} \vee x_n \vee \neg y_{n-3})$$

Donc le problème SAT est polynomialement réductible à 3-SAT. Donc 3-SAT est NP-complet.

Signalons d'autre part que tous les problèmes n-SAT avec $n \geq 3$ sont NP-complets, mais que le problème 2-SAT est lui P[20].

Le problème du sac à dos : Supposons que vous ayez un sac à dos susceptible de porter une charge m, et que vous disposiez d'un ensemble d'objets de poids p_i, $1 \leq i \leq n$. Votre problème est de montrer qu'il existe un sous-ensemble de ces objets tel que la somme de leur poids soit exactement égale à la capacité de votre sac. C'est le problème du sac à dos[21].

Mathématiquement, on peut exprimer la chose par :

$$\exists J, J \subset \{1, 2, \ldots, n\}, \sum_{i \in J} p_i = m$$

On peut démontrer que ce problème est NP-complet en montrant que 3-SAT lui est polynomialement réductible (voir (Salomaa 1989; Garey and Johnson 1979)).

Partition d'un ensemble de nombres : On considère un ensemble formé de n entiers positifs $A = \{a_1, \ldots, a_n\}$. On cherche à partager cet ensemble en deux sous-ensembles tels que les sommes des éléments de chacun des sous-ensembles soient égales. Mathématiquement, on cherche à montrer l'existence d'un sous-ensemble $J \subset A$ tel que :

$$\sum_{a_i \in J} a_i = \sum_{a_i \in A - J} a_i$$

On peut démontrer que ce problème est NP-complet en montrant que 3-SAT lui est polynomialement réductible.

---

20 On ne peut donc pas réduire polynomialement SAT à 2-SAT.

21 Signalons la désignation anglaise KNAPSACK que l'on trouve encore dans certaines traductions.

Existence d'un stable dans un graphe : Soit un graphe $G = (X, U)$, X représente l'ensemble des sommets du graphe et U l'ensemble des arcs.

On cherche à savoir s'il existe un sous-ensemble $X' \subset X$ de cardinal m tel que : $\forall x, y \in X', (x, y) \notin U$.

Ce problème est appelé *problème de recherche d'un stable* pour un graphe G.

On démontre que ce problème est NP-complet en prouvant que SAT lui est polynomialement réductible. Voici une idée de la démonstration. Pour une formule de SAT :

$$F = C_1 \wedge C_2 \wedge \ldots C_m$$

et

$$C_i = u_{j_i} \vee \ldots \vee u_{j_{k(i)}}$$

on construit le graphe $G = (V, E)$ (V pour « *vertex* », ensemble des sommets et E pour « *edges* », ensemble des arcs) défini par :

$$V = \{(i, u_j) | u_j \text{ apparaît dans la clause } C_i\}$$

$$E = \{((i, u_j), (i', u_{j'})) | (i = i') \vee (u_j = \neg u_{j'})\}$$

Le lecteur curieux pourra vérifier que F est satisfiable si (V, E) possède un stable de cardinal m et que la construction du graphe prend un temps polynomial.

Existence d'un transversal dans un graphe : Soit un graphe $G = (X, U)$. On dit que X' est un transversal de G si $X' \subset X$ et chaque arc de G a un sommet dans X'.

Le problème du transversal est trivialement équivalent à celui du stable. En effet, Y est un stable de $G = (X, U)$ ssi $X - Y$ est un transversal de G.

Existence d'un cycle hamiltonien dans un graphe : Exhiber un cycle hamiltonien de cardinal m dans un graphe G consiste à trouver une chaîne passant une et une seule fois par chaque sommet d'un sous-ensemble X' du graphe, avec X' de cardinal m.

On montre que le problème du transversal est polynomialement réductible à celui du cycle hamiltonien. La démonstration est relativement complexe.

Existence d'un circuit hamiltonien dans un graphe : Trouver un circuit hamiltonien de cardinal m consiste à trouver un chemin passant une et une seule fois par chaque sommet d'un sous-ensemble X' de X dans un graphe (X, U) orienté.

Le problème du circuit hamiltonien est polynomialement réductible à celui du cycle hamiltonien.

On commence par construire un graphe G' non orienté contenant 3 fois plus de points que G, car à chaque point de G on fait correspondre trois points x_1, x_2, x_3. Pour chaque point x de G, on connecte sur x_1 dans G' les arcs qui arrivent sur x dans G, on connecte sur x_3 les arcs qui partent de x dans G et on crée pour chaque triplet x_1, x_2, x_3 deux arcs : $(x_1, x_2), (x_2, x_3)$.

Le graphe ainsi construit est tel que l'existence d'un circuit hamiltonien dans G est équivalent à l'existence d'un cycle hamiltonien dans G'.

Le problème du voyageur de commerce : Ce problème[22] est un des plus célèbres problèmes de la théorie de la complexité. Nous le retrouverons lorsque nous parlerons des schémas d'approximation.

---

22 Le problème du voyageur de commerce a de nombreuses applications pratiques sous diverses formes : calcul de routage de circuits imprimés, calcul de réseaux de communication ou de distribution ...

Imaginons un voyageur de commerce. Il doit passer une et une seule fois dans chaque ville parmi un ensemble de n villes. Chaque ville i est séparée de la ville j par une distance d_{ij}. Quel chemin doit-il suivre afin de parcourir le moins de distance ?

Le problème du voyageur de commerce est un problème appartenant à la classe des problèmes d'optimisation. Un des problèmes de décision qui lui correspond consistera à montrer l'existence d'un circuit dont la longueur est inférieure ou égale à une constante k. Le problème est NP-complet.

On peut aisément le montrer en prouvant que le problème du cycle hamiltonien lui est polynomialement réductible. Pour ce faire, il suffit pour un graphe $G = (X, U)$ donné de construire le problème du voyageur de commerce en posant :
- $d_{ij} = 1$ si (i, j) est une arête de G
- $d_{ij} = 2$ si (i, j) n'est pas une arête de G.

S'il existe une solution au problème du voyageur de commerce pour $n = k$ alors on sait qu'il existe un cycle hamiltonien dans G de cardinal k et réciproquement.

Planification de tâches indépendantes : On dispose de m machines identiques qui ne peuvent exécuter qu'une seule tâche à la fois et l'on doit faire exécuter n tâches, chaque tâche i durant un temps t_i. Le problème consiste à montrer qu'il existe une affectation des tâches à chaque machine telle que le travail soit terminé avant un temps T fixé. Pour démontrer que ce problème est NP-complet, on peut démontrer que le problème du sac à dos lui est polynomialement réductible.

Il existe plusieurs variantes de ce problème. On peut ainsi exiger que certaines des tâches soient effectuées dans un certain ordre, ou que certaines machines ne puissent effectuer que certaines tâches. Toutes ces variantes sont NP-complètes.

Il faut toujours se méfier, lorsque l'on examine un problème, de trop se fier à une ressemblance. Ainsi, le problème consistant à montrer qu'il existe une chaîne simple reliant deux sommets donnés, et de longueur *inférieure* à k, dans un graphe dont les arêtes sont valuées positivement, est un problème polynomial. En revanche, l'existence d'une chaîne simple de longueur *supérieure* à k dans le même type de graphe définit un problème NP-complet. De même pour 2-SAT et 3-SAT. Les exemples de ce type sont nombreux. (Garey and Johnson 1979) présente un large catalogue de problèmes NP-complets (et autres).

11.4.5 Preuve de NP-complétude

Nous donnons ici deux exemples simples montrant comment démontrer le caractère NP-complet d'un problème à partir d'un problème connu comme étant NP-complet.

- Le premier type de preuve, dite par restriction, consiste à montrer qu'une restriction du problème considéré est déjà connue comme étant un problème NP-complet (il faudrait aussi montrer que le problème est dans NP). Considérons par exemple le problème du sac à dos :

$$\exists J, J \subset \{1, 2, \ldots, n\}, \sum_{i \in J} p_i = m$$

On peut se restreindre au cas où $m = \frac{1}{2}\sum_{i=1}^{n} p_i$. On a alors :

$$\sum_{i \in J} p_i = \frac{1}{2}\sum_{i=1}^{n} p_i$$

soit, en multipliant par 2 et éliminant à gauche et à droite les termes identiques :

$$\sum_{J} p_i = \sum_{\overline{J}} p_i$$

On reconnaît alors le problème de partition. Si l'on sait que ce problème est NP-complet, alors notre problème de sac à dos l'est également (s'il est dans NP).

– Le second type de preuve, dit par transformation polynomiale, a déjà été évoqué. Considérons à nouveau le problème du sac à dos, que nous supposerons NP-complet :

$$\exists J, J \subset \{1, 2, \ldots, n\}, \sum_{i \in J} p_i = m$$

On peut, à partir de ce problème considérer le problème de partition de l'ensemble A des p_i auquel on adjoint un $p_{n+1} = 2m - \sum_{i=1}^{n} p_i$:

$$\sum_{p_i \in J} p_i = \sum_{p_i \in A-J} p_i$$

Le terme p_{n+1} est soit dans J, soit dans $A - J$. On supposera, sans perte de généralité qu'il est dans $A - J$. On a alors :

$$\sum_{p_i \in J} p_i = \left(\sum_{p_i \in A-J-\{p_{n+1}\}} p_i\right) + 2m - \sum_{i=1}^{n} p_i$$

soit :

$$\sum_{p_i \in J} p_i = m$$

Ce problème a une solution ssi le problème de sac à dos initial a une solution. D'autre part, il est simple de montrer que le passage du problème de sac à dos initial au problème de partition final peut s'effectuer en un temps polynomial. On a donc exhibé une transformation polynomiale du premier problème dans le second : si le problème du sac à dos est NP-complet, le problème de partition l'est également.

11.5 Schémas d'approximation

Lorsque l'on sait qu'un des problèmes de décision associés à un problème d'optimisation est NP-complet, il n'y a aucune chance, dans l'état actuel des connaissances[23], de pouvoir résoudre ce problème d'optimisation, *dans le cas général*, de façon efficace.

---

23 Rappelons encore une fois qu'il n'est pas prouvé que les problèmes NP-complets n'aient pas de solution polynomiale.

On peut alors chercher des algorithmes qui, sans résoudre ce problème de façon optimale, garantissent de trouver une solution dans un temps polynomial et garantissent également que l'écart avec la solution optimale peut être majoré.

11.5.1 Principes généraux

Définition 11.14 – ε-approximation d'un problème Π – *Soit un problème Π résolu de façon optimale par l'algorithme OPT pour toute donnée x.*

L'algorithme A_ε est une ε-approximation résolvant Π ssi :

$$\forall x, \left| \frac{A_\varepsilon(x) - OPT(x)}{OPT(x)} \right| < \varepsilon$$

Définition 11.15 – Schéma d'approximation – *On dit que l'on possède un schéma d'approximation d'un problème Π si pour tout $\varepsilon > 0$, il existe un algorithme A_ε qui soit une ε-approximation de Π.*

Définition 11.16 – Schéma d'approximation polynomial – *Un schéma d'approximation est dit polynomial si pour ε fixé, le temps d'exécution de A_ε est borné par un polynôme $P(n)$ où n représente la taille des données.*

Définition 11.17 – Schéma d'approximation complet – *Un schéma d'approximation est dit complet si le temps d'exécution de A_ε est borné par un polynôme $P(n, 1/\varepsilon)$ où n représente la taille des données.*

Nous allons, dans la suite, nous intéresser au problème des ε-approximations. En particulier, nous allons constater qu'il existe des problèmes qui n'admettent pas d'ε-approximation polynomiale.

11.5.2 Le problème du voyageur de commerce

Théorème 11.5 – *Si* P $\neq$ NP, *le problème du voyageur de commerce n'admet aucune ε-approximation polynomiale, quel que soit ε positif.*

Nous allons démontrer complètement ce théorème. Pour ce faire, nous allons raisonner par l'absurde et démontrer que, s'il existe une ε-approximation polynomiale du problème du voyageur de commerce, alors le problème du cycle hamiltonien peut être résolu en temps polynomial.

Supposons donc qu'il existe un ε et un algorithme A_ε tel que A_ε soit une ε-approximation du problème du voyageur de commerce. Comme toute solution du voyageur de commerce ne peut être que supérieure à la solution optimale, nous avons :

$$OPT(x) \leq A_\varepsilon(x) \leq (1 + \varepsilon)OPT(x)$$

Considérons maintenant un graphe $G = (X, U)$, avec $X = (x_1, \ldots, x_n)$.

Nous allons prendre comme problème du voyageur de commerce :

- l'ensemble des villes est l'ensemble des sommets du graphe ;

- $d(x_i, x_j) = 1$ si (x_i, x_j) est une arête de G ;
- $d(x_i, x_j) = \lfloor n\varepsilon \rfloor + 2$ sinon. ($\lfloor x \rfloor$ désigne la partie entière de x).

On remarque immédiatement que si G a un cycle hamiltonien, alors la solution du problème du voyageur de commerce est n (qui correspond à la tournée passant par ce cycle).

Si G n'a pas de cycle hamiltonien alors la solution du problème du voyageur de commerce est au moins :

$$n - 1 + \lfloor n\varepsilon \rfloor + 2 = n + (1 + \lfloor n\varepsilon \rfloor) > n(1 + \varepsilon)$$

Or l'algorithme A_ε, appliqué à notre problème donne une tournée de longueur :

$$C \leq (1 + \varepsilon)n$$

si le problème a un cycle hamiltonien et

$$C > (1 + \varepsilon)n$$

sinon.

Donc on sait résoudre le problème du cycle hamiltonien en un temps polynomial, alors que ce problème est NP-complet. Donc, si P $\neq$ NP, le problème du voyageur de commerce n'a pas d'ε-approximation polynomiale.

Ce type de résultat est très important : il n'existe en fait aucune méthode polynomiale permettant de résoudre le problème du voyageur de commerce, ni même d'approcher la solution par un algorithme polynomial en majorant l'erreur.

11.5.3 Problèmes admettant des ε-approximations

Nous allons présenter ici deux problèmes admettant des ε-approximations.

Le problème du Δ-voyageur de commerce : Ce problème est le même que le problème du voyageur de commerce à la différence près que les distances entre les villes vérifient l'inégalité triangulaire :

$$d(i, j) \leq d(i, k) + d(k, j)$$

Il existe au moins deux algorithmes donnant des ε-approximations de ce problème. L'algorithme de Prim donne une approximation avec $\varepsilon = 1$ et réclame un nombre d'opérations de l'ordre de n^2. L'autre algorithme est basé sur la construction d'un cycle eulérien. Il réclame un nombre d'opérations de l'ordre de n^3 et l'on a $\varepsilon = 0.5$.

Planification de tâches indépendantes : Il existe un algorithme trivial qui est une ε-approximation de ce problème. Il consiste à affecter tous les temps $t_1, \ldots, t_n$, en commençant par t_1, systématiquement sur la machine la moins chargée jusque là.

On peut démontrer que si le nombre de machines est m, $\varepsilon = \frac{1}{3} - \frac{1}{3m}$.

	0	1	2	3	4	5	6	7	8	9	10
1	V	F	V	F	F	F	F	F	F	F	F
2	V	F	V	V	F	V	F	F	F	F	F
3	V	F	V	V	V	V	V	V	F	V	F
4	V	F	V	V	V	V	V	V	V	V	V
5	V	F	V	V	V	V	V	V	V	V	V

Table 11.1 – Exemple de résolution du problème de partition

11.6 Problèmes pseudo-polynomiaux

Nous allons, dans cette section, présenter rapidement et informellement une classe importante de problèmes reliés aux problèmes NP-complets, les problèmes pseudo-polynomiaux[24].

Prenons une instance du problème de partition[25], et examinons un algorithme permettant de le résoudre.

Soit $A = \{a_1 = 2, a_2 = 3, a_3 = 4, a_4 = 5, a_5 = 6\}$. Nous avons $n = 5$ éléments, et la somme B de l'ensemble des éléments est 20, donc, pour chacun des deux ensembles construits par partition, la somme ne peut être que $B/2 = 10$ Nous allons construire le tableau 11.1 ; en abscisse, nous portons $1 \leq i \leq B/2$, en ordonnée $1 \leq j \leq n$. Nous posons pour chaque case, $b_{i,j} = V$ *s'il existe un sous-ensemble* E *de* $\{a_1, \ldots, a_j\}$ *tel que la somme des éléments de* E *est* i. Sinon, nous posons $b_{i,j} = F$.

Le tableau est construit par l'algorithme suivant :

1. Pour la première ligne, nous avons $b_{i,1} = V$ si $i = 0$ ou $i = a_1$, et $b_{i,1} = F$ sinon.
2. Pour toutes les autres lignes j ($j > 1$), nous avons $b_{i,j} = V$ si :
   - $b_{i,j-1} = V$ ou si
   - $a_j \leq i$ et $b_{i-a_j,j-1} = V$

On sait que le problème de partition a une solution si $b_{B/2,n} = V$.

Si l'on considère attentivement l'algorithme, on remarque qu'il est polynomial en fonction du nombre de cases du tableau $\frac{nB}{2}$. Bien entendu, il n'est pas polynomial en $n \log B$ (taille du codage des données), et nous n'avons donc pas un algorithme polynomial au sens de la complexité. Cependant, si nous imposions une borne sur la taille des nombres utilisables, l'algorithme deviendrait polynomial, puisque $\log B$ serait borné. Il suffirait même d'avoir une borne pour les données qui soit une fonction polynomiale de la taille du problème, pour que notre algorithme reste polynomial. Tout problème vérifiant cette propriété est dit *pseudo-polynomial.*

Un problème NP-complet peut être pseudo-polynomial, ou ne pas l'être. Ainsi, le problème du voyageur de commerce n'est pas pseudo-polynomial, alors que le problème de planification de tâches indépendantes l'est. Un problème NP-complet qui n'est pas

---

24 Pour plus de détails sur les problèmes pseudo-polynomiaux et les problèmes numériques, voir (Garey and Johnson 1979).

25 Nous savons (section 11.4.4) que ce problème est NP-complet ; donc il n'existe pas d'algorithme polynomial permettant de le résoudre dans l'état actuel des connaissances.

pseudo-polynomial est dit *fortement* NP-complet ou NP-complet dans le sens fort. Tous les problèmes NP-complets ne faisant pas intervenir de nombres sont « naturellement » fortement NP-complets.

Sur un plan pratique, il est important de savoir si un problème est pseudo-polynomial. En effet, très souvent, la taille des données est effectivement bornée, et un algorithme pseudo-polynomial permet alors de résoudre le problème dans un temps polynomial. Par exemple, en ce qui concerne la planification de tâches indépendantes, il existe toujours, dans les cas pratiques, une borne supérieure sur la longueur des tâches.

11.7 La cryptographie : une application

L'utilisation des ordinateurs sur une grande échelle, le remplacement lent mais régulier du courrier classique par le courrier électronique, les problèmes de sécurité et de confidentialité des informations, ont donné à la cryptographie une place de plus en plus importante[26].

C'est en 1975 que Diffie et Hellman (Diffie and Hellman 1976) introduisirent la notion de codage à *clé publique*. Dans ce système, le destinataire d'un message publie la clé de codage, et garde secrète la clé de décodage. Toute personne souhaitant lui envoyer un message le code avec la clé publique du destinataire. Celui-ci n'a alors plus qu'à décoder avec sa clé (privée) de décodage le message reçu. Pour qu'un tel système soit possible, il faut trouver un mécanisme de construction de clés de codage et de décodage tel que la connaissance de la clé de codage ne permette pas de déduire la clé de décodage. La théorie de la calculabilité et la théorie de la complexité sont là d'une grande utilité, car le créateur de la clé doit essayer d'estimer quel est le coût, en temps de calcul, pour un éventuel « briseur de code ». Nous allons voir cela sur un cas pratique, un exemple du système KNAPSACK.

Dans ce système, le destinataire du message va choisir un vecteur de n éléments de $\mathbb{N}$: $A = (a_1, a_2, \ldots, a_n)$ tel que pour tout i, on ait $(\sum_{j<i} a_j) < a_i$. Puis il choisit un nombre M (le modulo) tel que $M > 2a_n$ et un nombre u, $u < M$ et premier avec M. Il construit maintenant le vecteur $B = (b_1, b_2, \ldots, b_n)$ avec $b_i \equiv ua_i\ [M]$. Il publie alors le vecteur B qui est la clé de codage publique. Tout émetteur voulant coder un message le transformera tout d'abord en une suite de 0 et 1 (en remplaçant, par exemple, chaque lettre par son numéro d'ordre dans l'alphabet en binaire). Puis il découpera sa suite de 0 et de 1 en p paquets de n bits chacun. Il codera le paquet k de n bits $(\delta_{1,k}, \ldots, \delta_{n,k})$ par le nombre :

$$1 \leq k \leq p,\ \ C_k = \sum_{1 \leq i \leq n} \delta_{i,k} b_i$$

Il ne lui reste plus qu'à diffuser les nombres C_k. Une personne malveillante qui souhaiterait décoder le message devrait, pour chaque nombre C_k, chercher les $\delta_{i,k}$. Il s'agit donc de résoudre le problème du *sac à dos*[27] tel que nous l'avons présenté précédemment. Or

---

26 Pour une présentation générale des systèmes cryptographiques, voir, par exemple, (Robling-Denning 1982). Pour les problèmes d'authentification, voir (Needham 1990).

27 Sac à dos = *knapsack* en anglais, d'où le nom du système de codage.

nous savons que ce problème est NP-complet dans le cas général. Si le vecteur B est long et si les nombres qui le composent sont grands, la théorie de la complexité nous montre que notre espion a fort peu de chance de réussir à décoder le message[28].

Maintenant, il reste au destinataire à décoder le dit message. Cela se fait aisément grâce à quelques bonnes propriétés de l'arithmétique. À partir d'un nombre C_k, le décodeur construit le nombre[29] $D_k \equiv u^{-1}C_k \ [M]$. Il lui suffit alors de résoudre le problème du sac à dos sur D_k avec A comme vecteur de nombres. Mais ce sous-problème est trivial à résoudre en raison de la forme du vecteur A. Comme chaque a_i vérifie la relation $a_i > (\sum_{j<i} a_j)$, il suffit, en commençant par $i = n$, de comparer a_i à D_k. Si $a_i > D_k$, on note $\delta_{i,k} = 0$ et on passe à a_{i-1}. Si $a_i \leq D_k$ on pose $D_k = D_k - a_i$, on note que $\delta_{i,k} = 1$ et on passe à a_{i-1}, et ceci jusqu'à $i = 1$. Le problème est alors résolu. La séquence de bits solution représente le paquet k de n bits de l'expéditeur, décodé.

Il existe de nombreux algorithmes à clé publique, le plus célèbre étant certainement le système RSA (Rivest, Shamir, Adleman). Pourtant, on ne connaît aucun système dont la sécurité soit garantie. Suivant l'expression attribuée à Adi Shamir, la cryptographie reste la lutte éternelle entre les créateurs de code et ceux qui tentent de les briser.

11.8 Complexité en terme d'espace

Nous avons jusqu'ici considéré uniquement le problème de la complexité au sens du temps de calcul. Rien n'empêche de se poser le même type de questions concernant non plus le temps de calcul, mais l'espace de calcul nécessaire à la résolution.

Nous ne citerons dans ce chapitre que quelques résultats. On peut, pour plus de renseignements, se reporter à (Garey and Johnson 1979; Salomaa 1989; Johnson 1990).

Comme nous l'avons fait pour le temps, on définit la classe des langages PSPACE acceptés par une machine de Turing déterministe dont l'espace de calcul[30] sera borné par un polynôme P. De même on définit la classe des langages NPSPACE qui sont acceptés par une machine de Turing non-déterministe dont l'espace de calcul sera borné par un polynôme P.

Il existe un premier résultat intéressant :

Théorème 11.6 – NPSPACE = PSPACE.

---

28 Il faudrait cependant être plus prudent. En fait, la théorie de la complexité nous dit que, dans le cas général, le problème du sac à dos est NP-complet. Mais le cas présent n'est pas exactement le cas général, car le vecteur B n'est pas un vecteur choisi de façon quelconque. D'autre part, nous avons p exemples du problème pour le même vecteur B. En fait, Adi Shamir a démontré en 1982 qu'il était possible de casser un système KNAPSACK du type présenté ci-dessus par des méthodes (polynomiales) d'algèbre linéaire. Voir (Shamir 1982; Salomaa 1989).

29 M et u étant premiers entre eux, u^{-1} existe et il est aisé de le calculer par le théorème d'Euclide (u^{-1} représente l'inverse de u modulo M).

30 On s'intéresse normalement au nombre de cases utilisées pendant le calcul *en sus* des cases contenant l'entrée (le codage du problème). Mais on peut négliger ces dernières car si l'espace occupé est de taille polynomiale en la taille de l'entrée, l'espace consommé ajouté à l'espace occupé par l'entrée est encore de taille polynomiale en la taille de l'entrée. Pour d'autres classes (LOGSPACE par exemple (espace logarithmique), qui est inclus dans P), l'espace effectivement consommé peut être précisément défini en rajoutant deux bandes (et deux têtes) à la machine classique à une bande. Une bande d'entrée en lecture seulement qui contient l'argument et une bande de sortie, en écriture seulement, qui contiendra le résultat. C'est l'espace consommé sur la bande lecture/écriture qui est considéré.

Ce résultat est fort différent de celui que nous avions pour les classes temporelles. Il est assez facile à démontrer, il suffit de prouver qu'une machine de Turing non-déterministe avec un espace borné polynomialement peut être simulée par une machine de Turing déterministe avec un espace borné polynomialement.

Signalons un second résultat :

Théorème 11.7 – $\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} = \mathsf{NPSPACE}$.

En effet, une machine de Turing (déterministe ou non) utilise un espace toujours inférieur au temps qu'elle utilise (une case au plus peut être lue/écrite par transition). On ne sait pas à l'heure actuelle si les inclusions sont strictes ou s'il s'agit d'égalités. Comme pour la classe NP, on peut définir la classe des problèmes PSPACE-complets. Ces problèmes sont au moins aussi (et *a priori* plus) difficiles que les problèmes NP-complets. Citons quelques exemples de problèmes PSPACE-complets, voire même PSPACE difficiles (au moins aussi difficiles que les problèmes PSPACE-complets) :

- **Modal Logic Provability** : Une formule A propositionnelle d'un système modal $S \in \{K, T, S_4\}$. La formule A est elle prouvable ? Le problème est PSPACE-complet. Rappelons que le problème de satisfiabilité dans S_5 est lui NP-complet (Ladner 1977; Garey and Johnson 1979).
- **Jeux** : Une position de jeu de dames américaines ou de Go (avec des règles convenablement étendues pour que le jeu soit défini sur un damier $n \times n$). Existe-t-il une stratégie gagnante pour le joueur qui a le trait ? Ces problèmes sont PSPACE durs ! Il faut d'ailleurs noter que, tel qu'il est énoncé, le problème n'a jamais été résolu, ni par un homme, ni par une machine, sur les tailles habituelles pour ces deux jeux et sur la position initiale. Il semblerait que le cas des dames américaines ne soit pas loin d'être traité (d'après J. Shaeffer, auteur du logiciel CHINOOK).

11.9 Autres classes

De très nombreuses autres classes de problèmes ont été définies, pour des problèmes ne se limitant pas aux problèmes de décision (problèmes de recherche, d'énumération, d'optimisation...).

On définit ainsi la classe des problèmes (ou langages) EXPTIME dont la complexité temporelle pour une machine déterministe est bornée par $2^{p(n)}$, pour un polynôme p, fonction de la longueur de l'entrée n. A la différence de la classe des problèmes *prouvés intraitables*, cette classe limite la complexité par le haut et non par le bas : un problème de P est dans EXPTIME, mais n'est pas intraitable.

Nous invitons le lecteur à se reporter à (Johnson 1990) pour un exposé complet sur ces classes. La figure 11.1 résume rapidement les relations entre les différentes classes que nous avons définies[31].

31 Ne sont pas présentes les classes des problèmes intraitables et indécidables qui ne sont pas des sur-ensembles des autres classes : il existe des problèmes décidables, ou de EXPTIME qui ne sont pas intraitables et tout problème de P est dans NP, dans PSPACE, dans EXPTIME et est décidable.

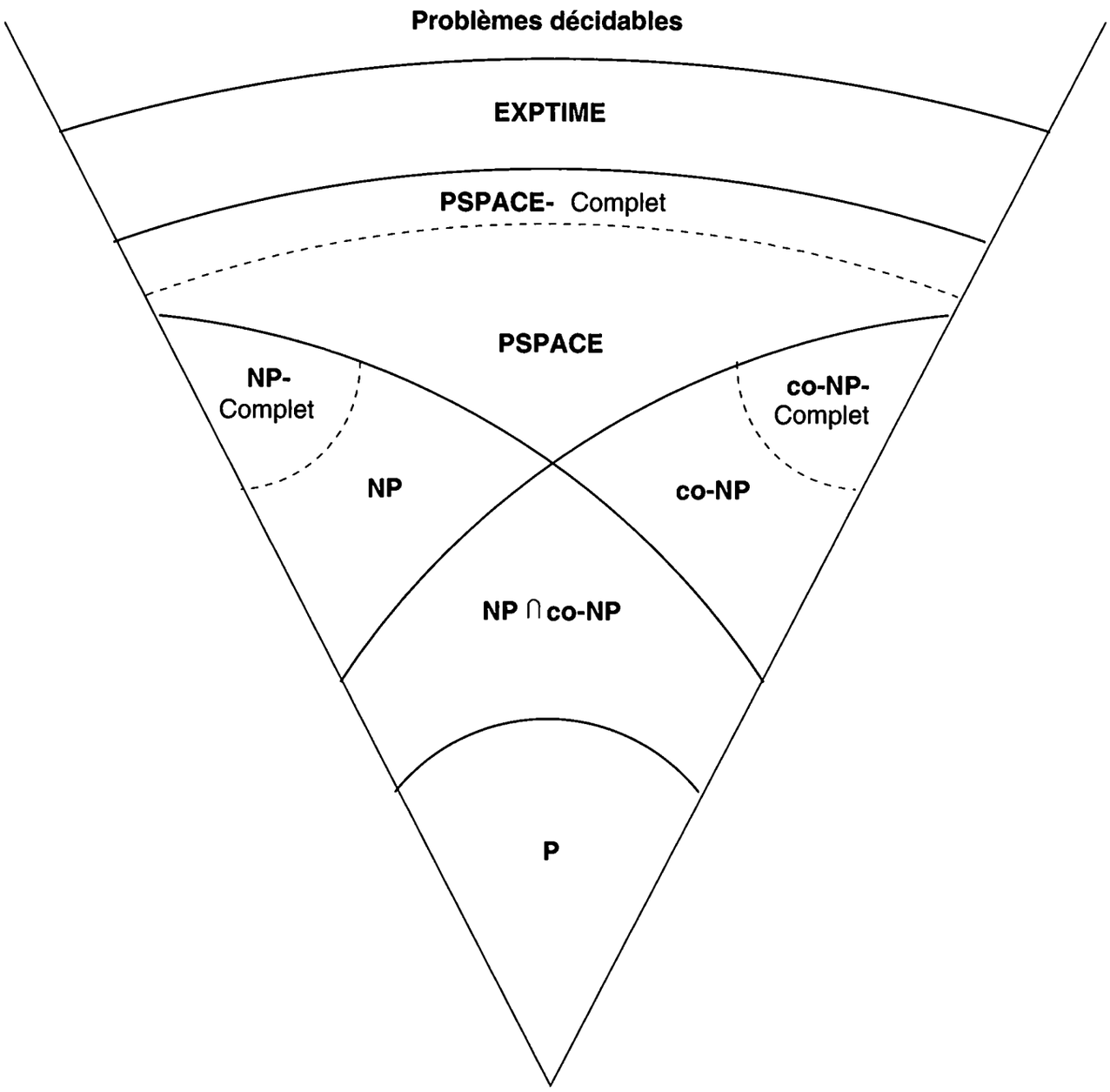


Figure 11.1 – Classes de complexité

CHAPITRE 12

λ-calcul

Thomas Schiex

12.1 Introduction

Le λ-calcul tend à formaliser la notion de *fonctions* telles que $f : x \mapsto x \times x$. L'approche se différencie de l'approche classique à deux titres au moins :

- alors que l'on ne fait habituellement pas de distinction (*a priori*) entre une définition en intension (ou compréhension) telle que $f(x) = 2x + 1$ et une définition en extension, formée de l'ensemble $\{\forall x \in D, (x, f(x))\}$, le λ-calcul ne s'intéresse qu'aux fonctions définies via des règles de calcul. Cette approche est plus adaptée à l'informatique qui est limitée au domaine du fini (ou tout au moins du dénombrable), et dont l'activité essentielle reste l'application de règles de calcul ;
- une fonction mathématique est habituellement caractérisée par son domaine et son co-domaine. Toutes les fonctions du λ-calcul ont un unique domaine et co-domaine : l'ensemble des termes du λ-calcul lui-même. Il est ainsi possible d'appliquer une fonction à elle-même, ce qui ne manquera pas de troubler au premier abord.

Créé dans les années 1930 par Alonzo Church, le λ-calcul constitue la base formelle (et le langage assembleur) de la programmation fonctionnelle, et des nombreux dialectes de celle-ci (LISP, SCHEME, ML, HOPE, MIRANDA...). Une des façons les plus simples de l'appréhender pour l'informaticien est de le considérer comme un langage de programmation élémentaire, mais contenant toute la puissance des langages fonctionnels : la « drosophile » des concepteurs de langages fonctionnels.

L'approche que nous allons suivre est identique à l'approche que nous avons suivie pour la présentation de la logique. Nous allons tout d'abord définir la structure du langage utilisé (définitions des termes du λ-calcul), puis nous étudierons les principes de réduction de ces formules (calcul). Nous ne ferons qu'évoquer informellement sa sémantique. Nous énoncerons les résultats théoriques essentiels concernant l'utilisation de ces règles de réduction. Nous étudierons enfin quelques exemples d'utilisation démontrant la puissance d'expression de ce formalisme bien frustre par ailleurs.

12.2 Définitions

Les termes du λ-calcul sont formés à partir des caractères « λ », « (» et «) », de variables x, y, z..., contenues dans un ensemble infini dénombrable Σ, et de symboles de constantes contenues dans un ensemble Γ (si Γ est vide, on parle de λ-calcul pur, sinon on parle de λ-calcul appliqué). Cette syntaxe est à l'origine des `lambda`-expressions de Lisp.

Nous dénoterons les termes par des lettres majuscules (K, S...) et les variables par des lettres minuscules ($x, y, z \ldots$). En général, deux lettres distinctes dénotent (sauf précision de notre part) deux variables distinctes. Nous utiliserons le symbole $\equiv$ pour dénoter l'égalité syntaxique de deux termes.

Définition 12.1 – *Un terme est obtenu en appliquant un nombre fini de fois les règles de formation suivantes :*

- **variable :** *Toute variable (élément de Σ) et tout symbole de constante (élément de Γ) est un terme du λ-calcul (appelé* atome*) ;*
- **application :** *Si A et B sont des termes du λ-calcul, (AB) est un terme du λ-calcul ;*
- **abstraction :** *Si x est une variable et A un terme du λ-calcul, alors $(\lambda x.A)$ est un terme du λ-calcul.*

Informellement, l'*abstraction* $(\lambda x.A)$ représente une fonction sans nom dont le paramètre formel est x et dont le corps est A. L'abstraction $(\lambda x.x)$ incarne donc, tout comme l'abstraction $(\lambda y.y)$, l'identité.

Le terme (AB) dénote lui l'application du terme A au terme B. Il faut noter qu'un même terme peut être utilisé aussi bien à gauche (en tant que fonction) qu'à droite (en tant qu'argument).

De façon traditionnelle, afin d'alléger l'écriture, on considère que l'application est associative à gauche et l'abstraction associative à droite. Il devient ainsi possible de simplifier :

$$((((A_1A_2)A_3)\ldots A_k) \rightsquigarrow A_1A_2A_3\ldots A_k \text{ et}$$

$$(\lambda x_1.(\lambda x_2.(\lambda x_3\ldots(\lambda x_k.A)))) \rightsquigarrow \lambda x_1.\lambda x_2.\lambda x_3\ldots x_k.A.$$

Enfin, $\lambda x.AB$ correspond à $(\lambda x.(AB))$ et non à $((\lambda x.A)B)$. Ces règles d'associativité inhabituelles forment une des premières difficultés du λ-calcul. Nous essaierons, bien que ce ne soit pas l'usage, de parenthèser complètement l'essentiel des expressions que nous présenterons afin d'en faciliter la lecture au « non-initié ». Exemples :

- $\lambda x.xx$ s'écrira $(\lambda x.(xx))$
- $ab(xy)(\lambda z.zx)$ s'écrira $(((ab)(xy))(\lambda z.(zx)))$
- $x(\lambda y.ab(cd))z$ s'écrira $((x(\lambda y.((ab)(cd))))z)$

N.B. : Une fonction de plusieurs variables, notée $\lambda x_1x_2x_3\ldots x_k.A$ s'exprime aisément au moyen de fonctions à un argument, telles que nos abstractions. On fait intervenir des *fonctions d'ordre supérieur i.e.*, des fonctions qui prennent en argument ou qui retournent en valeur d'autres fonctions. Le procédé permettant de passer de la fonction $f = (\lambda xy.A)$ d'arité 2 à la fonction $f' = (\lambda x.(\lambda y.A))$ d'arité 1 est appelé *curryfication*[1]. On se limitera donc aux fonctions à un argument sans perdre de généralité (au contraire

1 De H.B. Curry, un des pères du λ-calcul.

même puisque les fonctions curryfiées permettent, par exemple, l'application à une partie des arguments seulement).

Exemple : la fonction $\circ$ qui retourne la composée de deux fonctions d'arité 1 s'écrit $\circ = (\lambda fg.(\lambda x.(f(gx))))$. Une fois curryfiée, on obtient la fonction $\circ' = (\lambda f.\lambda g.\lambda x.(f(gx)))$. Cette nouvelle fonction de composition peut s'appliquer à un, deux ou trois arguments, pour retourner respectivement une fonction à deux, un ou sans arguments.

12.2.1 Variables libres ou liées

On définit l'ensemble des occurrences d'une variable x dans un terme A :

Définition 12.2 – *L'ensemble des occurrences d'une variable x dans un terme A se définit de façon inductive par :*

- *si A est la variable x, alors il s'agit d'une occurrence de x dans A ;*
- *si A est de la forme (BC), l'ensemble des occurrences de x dans A est formé par l'union des ensembles des occurrences de x dans B et dans C ;*
- *si A est de la forme $\lambda y.B$ ou $\lambda x.B$, l'ensemble des occurrences de x dans A est égal à l'ensemble des occurrences de x dans B.*

Cette définition s'étend aisément à l'occurrence d'un terme dans un autre terme.

Chaque occurrence d'une variable x dans un terme A du λ-calcul peut être *libre* ou *liée*.

Définition 12.3 – *L'occurrence d'une variable x est dite* liée *si et seulement si elle apparaît dans une partie B de A de la forme $\lambda x.C$. On dit alors que cette occurrence de x a pour portée $B \equiv \lambda x.C$. Elle est* libre *sinon.*

Du point de vue de l'informaticien, une occurrence liée est tout simplement l'occurrence d'un paramètre formel. La portée (ou « scope » en anglais) de l'occurrence liée n'est autre que la fonction F dont elle est le paramètre formel. La portée d'une variable est donc définie statiquement, en liaison avec le texte des fonctions, tout comme c'est le cas dans les langages à structure de bloc (PASCAL, C, ADA).

Définition 12.4 – *Toute variable ayant une occurrence libre (resp. liée) dans A est dite* variable libre *(resp.* liée*) dans A. Nous noterons $\mathcal{F}(A)$ l'ensemble des variables libres du terme A.*

Exemples :

- x est libre dans $(\lambda y.(\lambda z.((xy)z)))$,
- elle est liée dans $(\lambda x.(xx))$,
- elle est à la fois libre et liée dans $(\lambda y.(x(\lambda x.(yx))))$ (première occurrence libre, seconde occurrence liée).

Définition 12.5 – *On appelle* combinateur *ou* terme clos *un terme A qui ne contient pas de variable libre.*

Nous sommes maintenant syntaxiquement munis de l'essentiel des moyens d'expression d'un langage (fonctionnel) : définition et application de fonctions, notion de portée d'une variable. Il reste maintenant à définir des règles de calcul, purement syntaxiques elles-aussi, qui respectent le sens que nous donnons habituellement aux objets « fonctions », « portée », etc.

12.3 Calcul sur les termes du λ-calcul

12.3.1 Substitution

Il nous faut, pour définir la procédure de calcul sur les termes du λ-calcul, savoir comment substituer un terme à une variable libre dans un autre terme. Tout comme en logique du premier ordre, cette substitution ne se fait pas sans l'introduction de quelques définitions pas toujours très intuitives :

Définition 12.6 – *Étant donnés deux termes A et B et une variable x, on définit inductivement $[B/x]A$, la substitution de B à chaque occurrence libre de x dans A par :*

1. $[B/x]x = B$;
2. *Si A est un atome (variable ou symbole de constante) différent de x,* $[B/x]A = A$;
3. *Si A est de la forme* (CD), $[B/x](CD) = ([B/x]C[B/x]D)$;
4. *Si A est de la forme* $(\lambda x.C)$, *alors* $[B/x]A = A$;
5. *Si A est de la forme* $(\lambda y.C)$, *y différent de x, alors :*
   (a) *Si* $y \notin \mathcal{F}(B)$[2] *ou si* $x \notin \mathcal{F}(C)$, $[B/x](\lambda y.C) = (\lambda y.[B/x]C)$;
   (b) *Si* $y \in \mathcal{F}(B)$ *et si* $x \in \mathcal{F}(C)$, $[B/x](\lambda y.C) = (\lambda z.[B/x][z/y]C)$, *où z est une variable n'appartenant pas à* $\mathcal{F}((BC))$;

La clause (5b) permet de préserver le sens de $[B/x](\lambda y.C)$ indépendamment du statut (libre ou lié) de la variable y dans B.

Supposons cette clause (5b) absente, et considérons l'abstraction $\lambda y.x$, dont l'interprétation intuitive est « la fonction constante prenant comme valeur x en tout point ». Le terme $[z/x](\lambda y.x)$ devrait pouvoir être interprété intuitivement comme « la fonction constante prenant comme valeur z en tout point ». Effectivement, d'après la clause (5a), $[z/x](\lambda y.x) = (\lambda y.z)$.

Si nous considérons maintenant l'abstraction $\lambda z.x$, dont l'interprétation intuitive est naturellement la même que celle de $\lambda y.x$. On a, si l'on oublie la clause (5b), $[z/x]\lambda z.x = \lambda z.z$, qui n'est pas la fonction constante de valeur z, mais l'identité. Ce problème de collision de noms de variable (*name clash problem*) apparaît de la même façon en logique du premier ordre.

12.3.2 α-équivalence

L'égalité syntaxique de deux termes ($\equiv$) n'est pas très satisfaisante du point de vue « fonctionnel », puisqu'elle distingue par exemple les deux termes $(\lambda x.x)$ et $(\lambda y.y)$ qui représentent intuitivement tous deux la même fonction identité.

L'α-équivalence va nous permettre de disposer, de façon plus générale, d'une relation qui confond les termes ne se distinguant que par des noms de paramètres formels (variables liées) distincts, c'est-à-dire par renommage de variables :

Définition 12.7 – *Soient A un terme, x une variable liée dans A, de portée $B = (\lambda x.C)$ et $y \notin \mathcal{F}(C)$. Le fait de remplacer B par $(\lambda y.[y/x]C)$ dans A est appelé renommage de variable dans A.*

---

2 Rappelons au lecteur qu'il s'agit de l'ensemble des variables libres du terme B.

Considérons par exemple le renommage de la variable x dans le terme $A \equiv (\lambda x.\lambda y.(xy))$. La portée de l'unique occurrence de x dans A est le terme $B \equiv A \equiv (\lambda x.\lambda y.(xy))$ et on a donc (dans les termes de la définition précédente) $C \equiv \lambda y.(xy)$. La variable y n'appartenant visiblement pas à $\mathcal{F}(C)$, le nouveau terme $A' \equiv (\lambda y.[y/x](\lambda y.(xy)))$ est le résultat du renommage de x. Il faut noter que la substitution $[y/x](\lambda y.(xy))$ fait intervenir la clause (5b) et que l'on obtient finalement $A' \equiv (\lambda y.\lambda z.(yz))$.

Définition 12.8 – *Nous dirons qu'un terme A est α-équivalent à un terme D (noté $A \equiv_\alpha D$) si et seulement si D a été obtenu à partir de A après application d'un nombre fini, éventuellement nul, de renommage de variables.*

Deux termes α-équivalents ont la même longueur et les mêmes variables libres (puisqu'ils ne diffèrent que sur le nom de certaines variables liées). On démontre de plus que la relation $\equiv_\alpha$, comme son nom pouvait laisser l'entendre, est une relation d'équivalence qui définit un ensemble quotient Λ de classes d'équivalence (chaque classe regroupant l'ensemble des termes du λ-calcul qui sont α-équivalents entre eux).

Exemple : $(\lambda x.(\lambda y.(z(xy))))$ est α-équivalent à $(\lambda u.(\lambda v.(z(uv))))$, puisqu'il résulte de l'application de deux renommages de variables liées (x et y). Ces deux termes seront donc dans la même classe d'équivalence. En revanche, l'expression $(\lambda x.(\lambda y.(z(xy))))$ n'est pas α-équivalent à $(\lambda x.(\lambda y.(w(xy))))$, car w n'est pas une variable liée.

12.3.3 β-réduction

La β-réduction constitue une modélisation du mécanisme de « passage de paramètre ». Dans un langage mathématique, la valeur d'une fonction définie en intension (telle que $f(x) = x + x$) en un point donné a est obtenue en substituant la valeur a à toutes les occurrences du paramètre de la fonction. On obtient ainsi $f(2) = [2/x](x + x) = 2 + 2$.

La β-réduction pourra s'appliquer à chaque fois qu'une *abstraction* (fonction) se verra présenter un argument dans une *application*.

Définition 12.9 – *On appelle* β-redex *tout terme A de la forme $((\lambda x.B)C)$. Ce terme peut se contracter pour former le terme correspondant $[C/x]B$, appelé* contractum.

Ce procédé de contraction[3] peut s'appliquer à tout terme A en remplacant un des β-redex qu'il contient par son contractum permettant d'aboutir au terme A'. On dit alors que A se β-contracte en A' (noté $A \triangleright_\beta^1 A'$). On dira que A se β-réduit en B ($A \triangleright_\beta B$) si et seulement si B résulte de l'application d'un nombre fini (éventuellement nul) de β-contractions à A.

Exemples :

---

3 L'ensemble des idées et des résultats présentés se retrouvent dans la théorie – très proche du λ-calcul – de la logique combinatoire. En logique combinatoire, les termes sont formés à partir de variables, de deux symboles de constantes appelés traditionnellement S et K et d'une règle d'application d'un terme à un autre. Il est possible de faire apparaître une correspondance entre termes et calcul en logique combinatoire et termes et calcul en λ-calcul en interprétant les symboles de constantes S et K comme les combinateurs $S = (\lambda x.(\lambda y.(\lambda z.((xz)(yz)))))$ et $K = (\lambda x.(\lambda y.x))$. L'avantage majeur de la logique combinatoire est l'absence de variables liées qui fait disparaître les différents problèmes techniques dans la substitution, et l'α-équivalence. Nous avons préféré présenter le λ-calcul qui possède une signification intuitive plus claire que celle de la logique combinatoire.

$$(((\lambda x.(\lambda y.x))a)b) \rhd_\beta^1 ((\lambda y.a)b) \text{ (contraction du } \beta\text{-redex } ((\lambda x.(\lambda y.x))a)$$
$$\rhd_\beta^1 a \qquad \text{(contraction du } \beta\text{-redex } ((\lambda y.a)b)$$

$$((\lambda x.(xx))(\lambda x.(xx))) \rhd_\beta^1 ((\lambda x.(xx))(\lambda x.(xx)))$$
$$\rhd_\beta^1 ((\lambda x.(xx))(\lambda x.(xx)))$$
$$etc.$$

Ce dernier exemple montre bien que la β-contraction ne simplifie pas forcément le terme traité[4], et qu'elle peut s'appliquer indéfiniment (le contractum contenant toujours au moins un β-redex).

12.3.4 Formes normales

Le procédé de calcul via β-réductions ne s'achève que lorsque la forme obtenue ne contient plus aucun β-redex. On peut considérer qu'il s'agit alors de la valeur finale de l'expression calculée.

Définition 12.10 – *Un terme A qui ne contient aucun β-redex est dit en forme normale (ou β-forme normale). Si un terme A se β-réduit en B qui est en β-forme normale, on dit alors que B est une β-forme normale de A.*

Exemples : Soit le terme $(\lambda x.(\lambda y.x))ab$, alors a est sa forme normale. Si l'on considère le terme $(\underline{((\lambda x.(\lambda y.x))a)}\ \underline{((\lambda x.(xx))(\lambda x.(xx)))})$, il contient deux β-redex (soulignés) et peut donc se réduire de diverses façons :

1. en commençant par le β-redex de gauche :
$$(\lambda x.(\lambda y.x))a((\lambda x.(xx))(\lambda x.(xx))) \rhd_\beta^1 (\lambda y.a)((\lambda x.(xx))(\lambda x.(xx)))$$
$$\rhd_\beta^1 a$$

2. en réduisant $((\lambda x.(xx))(\lambda x.(xx)))$ à chaque étape :
$$(\lambda x.(\lambda y.x))a((\lambda x.(xx))(\lambda x.(xx))) \rhd_\beta^1 ((\lambda x.(\lambda y.x))a((\lambda x.(xx))(\lambda x.(xx)))$$
$$\rhd_\beta^1 ((\lambda x.(\lambda y.x))a((\lambda x.(xx))(\lambda x.(xx)))$$
$$etc.$$

On constate donc :
- qu'un terme ne possède pas forcément une forme normale (le terme $\Omega = ((\lambda x.(xx))(\lambda x.(xx)))$ n'en possède pas puisqu'il ne contient qu'un β-redex, et se réduit en lui-même) ;
- que de plus la façon dont s'opère la réduction semble d'importance quant à la forme obtenue (quand elle existe).

Si la β-réduction est censée modéliser un calcul et si, quand une forme normale existe, elle représente la valeur finale obtenue, il faut que la forme normale obtenue soit indépendante de l'ordre de réduction, sans quoi une même forme pourrait avoir plusieurs « valeurs » distinctes.

---

4 Elle peut même faire le contraire : considérez le terme $((\lambda x.((xx)x))(\lambda x.((xx)x)))$.

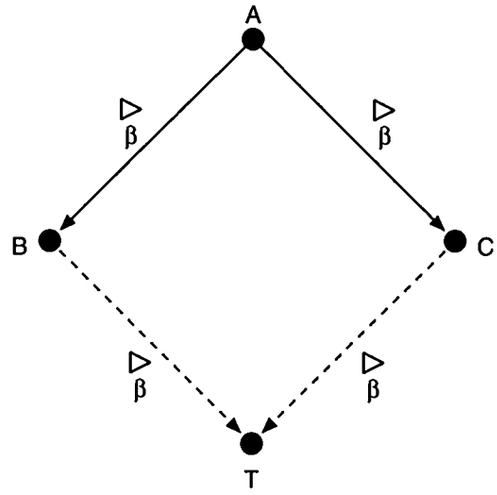


Figure 12.1 – Propriété du diamant

12.3.5 Théorème de Church-Rosser

Le théorème de Church-Rosser (propriété du diamant) pour la β-réduction (démonstration dans (Krivine 1990; Lalement 1990; Hindley and Seldin 1986; Barendregt 1981) par exemple) prouve que la forme normale, quand elle existe, est unique aux renommages de variables près :

Théorème 12.1 – Propriété de Church-Rosser – *Si* $A \triangleright_\beta B$ *et si* $A \triangleright_\beta C$, *alors il existe un terme* T *tel que* $B \triangleright_\beta T$ *et* $C \triangleright_\beta T$.

Le nom de *propriété du diamant* (*diamond property*) découle de la figure 12.1. On parle plus généralement de propriété de *confluence* des règles de réduction.

On déduit du théorème précédent que si A a deux β-formes normales B et C, alors $B \equiv_\alpha C$. En effet, on sait qu'il existe un T tel que B et C se réduisent à T. Or, B et C sont tous deux en β-forme normale, et ne contiennent donc pas de β-redex, ils sont donc forcément α-équivalents.

Dans le cadre du λ-calcul appliqué, on introduit parfois, parallèlement aux symboles de constantes, des règles de réduction sur ces symboles, permettant de leur donner la sémantique que l'on désire. Ces règles, dites règles de δ-réduction peuvent remettre en cause le résultat de confluence obtenu dans le cadre du λ-calcul pur.

Exemple : On désire rajouter au λ-calcul la notion de non-déterminisme. On rajoute pour cela le symbole de constante `or` et les règles de formation et de δ-réduction suivantes :

- *Formation :* si M et N sont deux termes, alors $(M \texttt{ or } N)$ est un terme ;
- *δ-réduction :* un terme de la forme $(M \texttt{ or } N)$ peut se réduire en M ;
- *δ-réduction :* un terme de la forme $(M \texttt{ or } N)$ peut se réduire en N ;

Le système obtenu est non-confluent puisque $x \texttt{ or } y$ peut se réduire en x ou en y qui ne peuvent plus se réduire à un même terme.

12.3.6 Ordre de réduction

Le théorème de Church-Rosser permet de ne pas distinguer a priori deux séquences de réductions qui aboutissent à deux formes normales. Mais que penser de termes dont

certaines réductions aboutissent à une forme normale, alors que d'autres ne se terminent pas ? (cf. exemple section 12.3.4).

Théorème 12.2 – Théorème de standardisation – *Si un terme A admet une β-forme normale, alors il existe une stratégie de réduction (dite réduction standard ou* leftmost outermost reduction*) qui permet d'atteindre cette forme normale*[5].

La *leftmost outermost reduction*, que l'on pourrait traduire par *réduction maximale gauche* consiste à réduire d'abord, parmi les différents β-redex d'un terme, le β-redex maximal situé le plus à gauche :

Définition 12.11 – β-redex maximal – *Étant donné un terme M, un β-redex contenu dans M est maximal ssi il n'est contenu dans aucun autre β-redex apparaissant dans M.*

L'exemple de réduction du terme $(\underline{(((\lambda x.(\lambda y.x))a)} \ \underline{((\lambda x.(xx))(\lambda x.(xx)))})$ illustre bien ce théorème. Ce terme contient deux β-redex (soulignés), tous les deux maximaux. La réduction par le β-redex de gauche *i.e.*, $((\lambda x.(\lambda y.x))a)$ aboutit à la forme normale, alors que la réduction par le β-redex de droite ne s'achève pas.

Cette propriété a une relation directe avec les stratégies d'évaluation des langages fonctionnels. La stratégie de réduction *leftmost outermost redex*, la plus sûre puisqu'elle garantit l'arrivée à une forme normale si elle existe, correspond à la stratégie *d'appel par nom* (*call by name* : on applique la fonction d'abord, on réduit les arguments en dernier lieu), rarement utilisée du fait de son inefficacité. En effet, si l'on considère la fonction $(\lambda x.(xx))$ et son application à un terme L (dont la réduction dans sa forme normale L' peut être très longue), on a, en utilisant la stratégie standard :

$$((\lambda x.(xx))L) \triangleright_\beta^1 (LL) \triangleright_\beta (L'L) \triangleright_\beta (L'L')$$

qui va donc entraîner deux réductions de L, alors qu'une stratégie opposée, qui correspond à la stratégie la plus couramment employée *d'appel par valeur* (*call by value*) va commencer par réduire l'argument L en L', une fois pour toutes (on évalue les arguments d'abord, on applique la fonction ensuite) :

$$((\lambda x.(xx))L) \triangleright_\beta^1 ((\lambda x.(xx))L') \triangleright_\beta^1 (L'L')$$

La meilleure solution consiste alors à apporter un raffinement technique à la réduction standard, consistant (grossièrement) à mémoriser le résultat de la réduction de L lorsqu'elle a lieu une première fois, pour le réutiliser directement lors des réductions suivantes. On parle alors *d'appel par besoin* (*call by need*) et *d'évaluation paresseuse* (*lazy evaluation*). Elle correspond, informellement, à la séquence :

$$((\lambda x.(xx))L) \triangleright_\beta^1 (LL) \triangleright_\beta (L'L')$$

dans laquelle tous les termes L ont été réduits en même temps[6].

---

5 Preuve dans (Hindley and Seldin 1986; Barendregt 1981) par exemple.

6 Lors de la substitution de L à x, on ne conserve qu'un exemplaire de L, partagé par ses deux occurrences.

12.3.7 β-équivalence

La β-réduction est non symétrique. Cependant, il est possible de définir la *β-équivalence* (ou *β-égalité*, notée $\equiv_\beta$) en posant qu'un terme A est β-égal à un terme B si et seulement si B est obtenu à partir de A après un nombre fini (éventuellement nul) de β-contractions, de β-contractions inversées[7] et de renommages de variables. La β-équivalence constitue un affaiblissement de l'α-équivalence qui va nous permettre de mettre en relation deux termes se réduisant en deux formes normales α-équivalentes.

Définition 12.12 – *Deux termes A et B sont β-équivalents (noté $A \equiv_\beta B$) si et seulement s'il existe une séquence de termes $A_0 \ldots, A_n$ telle que :*

$$A_0 \equiv A,$$

$$A_n \equiv B,$$

$$\forall i \leq (n-1), (A_i \rhd_\beta^1 A_{i+1}) \text{ ou } (A_{i+1} \rhd_\beta^1 A_i) \text{ ou } (A_i \equiv_\alpha A_{i+1})$$

De la même façon que pour la β-réduction, on a :

Théorème 12.3 – Propriété de Church-Rosser – *Soient deux termes A et B, tels que $A \equiv_\beta B$, alors il existe un terme T tel que $A \rhd_\beta T$ et $B \rhd_\beta T$.*

Ce théorème montre que deux termes β-égaux représentent intuitivement la même fonction, puisqu'ils peuvent tous deux être réduits au même terme. Dans la pratique, la relation de β-équivalence est indécidable[8], elle est cependant employée dans certains dialectes PROLOG réalisant une β-unification (λ-Prolog par exemple).

On en déduit aisément un certain nombre de corollaires, en particulier que si deux termes sont β-égaux, ils ont :
- soit tous les deux une forme normale ;
- soit aucun d'entre eux n'en possède une.

De plus, s'ils en possèdent chacun une, elles sont α-équivalentes.

12.3.8 Extensionnalité et η-réduction

Le concept d'égalité de deux fonctions mathématiques est habituellement *extensionnel i.e.*, si deux fonctions prennent la même valeur en tout point ($F = x + x$ et $G = 2 * x$) alors elles sont égales :

$$\forall x, f(x) = g(x) \Rightarrow f = g$$

Le λ-calcul muni des règles de β-réduction et d'α-équivalence ne présente pas cette propriété puisqu'il est possible d'exhiber des termes A et B tels que :

---

7 L résulte de la β-contraction inversée de M ssi $L \rhd_\beta^1 M$.

8 Plus précisément semi-décidable, puisque si deux termes ont une forme normale, on peut y aboutir par une réduction standard. Il s'agit du premier théorème d'indécidabilité de l'histoire des mathématiques. C'est à partir de lui que Church démontra l'indécidabilité du calcul des prédicats en 1936.

$$\forall x, (Ax) \equiv_\beta (Bx)$$

sans que l'on ait pour autant $A \equiv_\beta B$. Il suffit de considérer par exemple les termes $A \equiv y$ et $B \equiv (\lambda z.(yz))$.

Cette propriété d'extensionnalité peut être obtenue en rajoutant aux précédentes règles de réduction différents ensembles de règles. Le système le plus simple est obtenu en rajoutant une nouvelle règle dite d'η-réduction. L'exemple précédent montre qu'il est nécessaire de pouvoir faire disparaître des abstractions inutiles (disparition du paramètre z de l'abstraction $B \equiv (\lambda z.(yz))$.

Définition 12.13 – *On appelle η-redex tout terme de la forme $(\lambda x.(Ux))$ dans lequel $x \notin \mathcal{F}(U)$. Son contractum est alors le terme U.*

De la même façon que pour la β-réduction, on parle d'η-contraction (remplacement d'un η-redex par son contractum) ou d'η-réductions (plusieurs η-contractions). On les note respectivement $\rhd_\eta^1$ et $\rhd_\eta$.

Exemple : le terme $B \equiv (\lambda z.(yz)) \rhd_\eta^1 y$.

Il est ainsi possible de définir une version « étendue » du redex, incluant β-redex et η-redex.

Définition 12.14 – *On appelle $\beta\eta$-redex tout terme qui est un β-redex ou un η-redex. Son contractum est obtenu par application d'une β-contraction (si c'est un β-redex) ou d'une η-contraction (si c'est un η-redex).*

La substitution de la notion de $\beta\eta$-redex à la notion de β-redex dans les chapitres précédents induit la définition des $\beta\eta$-réduction, $\beta\eta$-équivalence, $\beta\eta$-formes normales, etc. Le système obtenu présente toutes les bonnes propriétés déjà exposées dans le cadre de la seule β-réduction (propriété de Church-Rosser, théorème de standardisation).

12.4 Expression de fonctions récursives

Nous nous sommes jusqu'ici intéressés à la description de fonctions non récursives. Dans les langages traditionnels, la possibilité d'exprimer des textes de fonctions récursives repose habituellement sur la facilité qui est offerte de **nommer** les fonctions, facilité absente du λ-calcul. Ces noms peuvent alors être utilisés à l'intérieur des fonctions, permettant d'exprimer des références mutuelles ou propres.

De façon plus précise, une fonction récursive F est définie en fonction d'elle-même et est donc solution d'une équation de la forme :

$$F \equiv_\beta (G\ F)$$

Si nous nous plaçons en λ-calcul appliqué, nous pouvons introduire les symboles de constantes $*, -, =, \texttt{if}, 0, 1$ auxquels on adjoint les règles de réduction adéquates[9] permettant de leur donner leur sémantique habituelle de multiplication, soustraction, égalité

9 Dites de δ-réduction. Ces règles peuvent rendre la réduction impossible et remettre en cause la confluence de la réduction par exemple.

numérique, conditionnelle. Nous pouvons ainsi exprimer que le terme correspondant à la fonction récursive « factorielle »(notée F) est solution de l'équation :

$$(\mathbf{F}\ n) \equiv_\beta (\texttt{if}\ (=\ n\ 0)\ 1\ (*\ n\ (\mathbf{F}\ (-\ n\ 1))))$$

soit :

$$\mathbf{F} \equiv_\beta (G\ \mathbf{F})$$

avec

$$G \equiv (\lambda f.(\lambda n.(\texttt{if}\ (=\ n\ 0)\ 1\ (*\ n\ (f\ (-\ n\ 1))))))$$

Un terme correspondant à une fonction récursive F est donc un point fixe d'une fonction d'ordre supérieur G *i.e.*, $F = (GF)$. Le problème d'expression d'une fonction récursive est alors ramené à la résolution d'une équation au point fixe sur les termes du λ-calcul.

12.4.1 Les combinateurs de point fixe

Dans le monde du λ-calcul, chaque terme possède un point fixe. Ainsi, pour tout terme F du λ-calcul, il existe un terme P, invariant par application de F :

$$(FP) \equiv_\beta P$$

Plus simplement encore, il existe un combinateur Y (un terme du λ-calcul sans variable libre) qui permet de trouver un de ces points fixes, c'est-à-dire Y est tel que, pour tout terme F du λ-calcul, (YF) est un point fixe de F.

Théorème 12.4 – Théorème du point fixe – *Il existe un combinateur Y tel que, pour tout terme F :*

$$(YF) \equiv_\beta (F(YF))$$

Il suffit de considérer le terme $Y_{Curry} = (\lambda f.((\lambda y.f(yy))(\lambda y.f(yy))))$
On a, pour tout terme F :

$$\begin{aligned}(Y_{Curry}F) &\equiv_\alpha (\lambda f.((\lambda y.f(yy))(\lambda y.f(yy))))F \\ &\rhd^1_\beta (\lambda y.(F(yy)))(\lambda y.(F(yy))) \\ &\rhd^1_\beta F((\lambda y.(F(yy)))(\lambda y.(F(yy)))) \\ &\equiv_\beta F(YF)\end{aligned}$$

Q.E.D.

On dit alors que Y est un combinateur de point fixe (*fixed-point combinator* ou *paradoxical combinator*[10]). Il faut noter que le nom de Y est généralement réservé aux combinateurs de point fixe. Dans la suite de notre exposé, toute apparition d'un terme Y correspondra effectivement à un combinateur de point fixe.

---

10 Un des « paradoxical combinators » les plus remarquables, esthétiquement au moins, est le suivant (par J.W. Klop). Il résulte d'une « sophistication » de Y_{Turing} :

$$Y_{Klop} \equiv LLLLLLLLLLLLLLLLLLLLLLLLLL$$

Théorème 12.5 – *Il est possible d'exhiber des combinateurs de point fixe ayant la propriété plus forte :*

$$(YF) \rhd_\beta (F(YF))$$

Preuve : on peut considérer par exemple le terme :

$$Y_{Turing} = ((\lambda z.\lambda x.x(zzx))(\lambda z.\lambda x.x(zzx)))$$

On a, pour tout terme V :

$$\begin{aligned}(Y_{Turing}V) &\equiv_\alpha ((\lambda z.\lambda x.x(zzx))(\lambda z.\lambda x.x(zzx)))V \\ &\rhd^1_\beta (\lambda x.x((\lambda z.\lambda x.x(zzx))(\lambda z.\lambda x.x(zzx)))x)V \\ &\rhd^1_\beta V((\lambda z.\lambda x.x(zzx))(\lambda z.\lambda x.x(zzx))V) \\ &\equiv_\alpha V(Y_{Turing}V)\end{aligned}$$

Q.E.D.

Les différents combinateurs permettent alors de résoudre, non seulement les équations au point fixe définissant une fonction récursive, mais plus généralement, toute équation en x de la forme $xy_1 \ldots y_n = F$ (où F est un terme pouvant contenir une occurrence de x).

Théorème 12.6 – *Pour tout terme F et $\forall n \geq 0$, l'équation $xy_1 \ldots y_n = F$ admet une solution en x* i.e., *il existe un terme X tel que $Xy_1 \ldots y_n \equiv_\beta [X/x]F$.*

Preuve : Il suffit de considérer $X = Y(\lambda x.\lambda y_1.\ldots.\lambda y_n.F)$ (où Y est un des combinateurs de point fixe) □.

Le terme correspondant à la fonction « factorielle » est solution de l'équation :

$$(\mathbf{F}n) \equiv_\beta ((G\mathbf{F})n)$$

avec

$$G \equiv (\lambda f.(\lambda n.(\texttt{if } (= \ n\ 0)\ 1\ (*\ n\ (f\ (-\ n\ 1))))))$$

Il sera défini[11] par :

$$\mathbf{F} \equiv (Y(\lambda f.(\lambda n.(\texttt{if } (= \ n\ 0)\ 1\ (*\ n\ (f\ (-\ n\ 1)))))))$$

L'utilisation de ce théorème ne se limite pas aux fonctions récursives, il est ainsi possible de définir (par exemple) le terme P (Poubelle) qui avale tous les arguments qui lui sont présentés par $(Px) \equiv_\beta P$, soit $P \equiv (Y(\lambda p.\lambda x.p))$.

---

où L est le terme :

$$L \equiv \lambda abcdefghijklmnopqstuvwxyzr.(r(this\ is\ a\ fixed\ point\ combinator))$$

Le lecteur pourra exhiber, en s'inspirant du combinateur précédent (qui n'est pas si compliqué qu'il y paraît), une version française de son choix.

11 La définition d'un terme par une équation de la forme $F = (GF)$ pose le problème de savoir ce qui a été précisément défini par cette équation. Dire que F est le point fixe de $G = (\lambda x.x)$ ne définit pas F (tous les termes vérifient cette équation). On convient de choisir la fonction définie par $F = (Y\ G)$.

12.5 Utilisation du λ-calcul

Le λ-calcul pur semble a priori trop frustre pour représenter la majorité des fonctions que nous utilisons couramment. En fait, Church a démontré que l'ensemble des fonctions λ-calculables et l'ensemble des fonctions Turing-calculables (via une machine de Turing) étaient identiques. La thèse de Church-Turing identifie alors fonctions λ-calculables, Turing-calculables et intuitivement calculables. La puissance d'expression du λ-calcul est donc aussi vaste que l'on pourrait le souhaiter.

Nous nous proposons d'exposer quelques-unes des modélisations les plus classiques reproduisant quelques objets familiers de l'informaticien. C'est toujours l'aspect calculatoire des objets qui sera utilisé pour la modélisation, les nombres effectuant itérativement des applications, les booléens effectuant des « branchements ».

12.5.1 Les booléens et opérations booléennes

L'utilisation la plus courante de booléens se fait dans la conditionnelle :

`(if <booléen> A B)`

qui consiste à sélectionner, suivant la valeur du `<booléen>` l'expression `A` ou `B`. Ceci peut nous amener à considérer les booléens comme des fonctions à deux arguments, le booléen VRAI (noté V) retournant son premier argument, FAUX (noté F) retournant le second :

$$V \equiv \lambda x.\lambda y.x$$

$$F \equiv \lambda x.\lambda y.y$$

L'opérateur conditionnel IF (noté C pour « Conditionnelle ») consiste alors tout simplement à appliquer le booléen aux deux termes arguments :

$$C \equiv \lambda b.\lambda x.\lambda y.bxy$$

On obtient, par exemple :

$$\begin{aligned}(C\ F\ (\lambda x.\lambda y.y)(\lambda x.\lambda y.x)) &\equiv ((\lambda b.\lambda x.\lambda y.bxy)(\lambda x.\lambda y.y)(\lambda x.\lambda y.y)(\lambda x.\lambda y.x)) \\ &\rhd_\beta ((\lambda x.\lambda y.y)(\lambda x.\lambda y.y)(\lambda x.\lambda y.x)) \\ &\rhd_\beta (\lambda x.\lambda y.x) \\ &\equiv V\end{aligned}$$

Il est alors possible de définir les opérateurs de négation (noté N), de conjonction (noté E pour ET) et de disjonction (noté O pour OU) en s'appuyant sur les propriétés classiques de ces opérateurs :

- si x est vrai, sa négation est égale à faux, et vice-versa, donc :

$$(Nx) \equiv (CxFV)$$

- la disjonction de x et de y est vraie si x est égal à vrai, sinon elle est égale à la valeur de y. On en déduit une écriture possible de O :

$$O \equiv \lambda x.\lambda y.xVy$$

– la conjonction de x et de y est égale à faux si x est égal à faux, elle est égale à la valeur de y sinon. On en déduit une écriture possible pour E :

$$E \equiv \lambda x.\lambda y.xyF$$

Tout comme les définitions qui suivent et qui précèdent, ces exemples peuvent être codés dans tout langage fonctionnel (SCHEME, HOPE, MIRANDA...)[12].

12.5.2 Les doublets

Un doublet est habituellement considéré comme une structure de donnée passive, constituée de deux champs appelés (pour des raisons historiques) champs CAR (ou tête) et CDR[13] (ou queue). Un doublet est retourné par l'appel à une fonction de construction (`CONS`) à deux arguments qui seront les contenus des champs CAR et CDR du doublet créé. Une fois construit, il est possible d'accéder au contenu des champs CAR et CDR d'un doublet en y appliquant des fonctions de sélection (habituellement notées `CAR` ou `FIRST` et `CDR` ou `REST`).

La représentation des doublets en λ-calcul contient un peu des idées des langages dits « objets ». Un doublet sera un objet, contenant localement la valeur de ses deux champs (CAR et CDR) et qui peut recevoir deux types de messages (des méthodes de sélection des champs, bien sûr) *i.e.*, CAR (ou tête) et CDR (ou queue).

L'opérateur CONS (noté D pour doublet) doit construire un doublet (formé d'une tête et d'une queue). Il va pour cela retourner une fonction qui prendra en argument une fonction de sélection s qui sera appliquée à tête et queue. On aboutit donc à :

$$D \equiv \lambda t.\lambda q.\lambda s.stq$$

La création d'un doublet se fera simplement en appliquant le terme D à deux arguments quelconques a et b :

$$\begin{aligned}(D\ a\ b) &\equiv ((\lambda t.\lambda q.\lambda s.stq)ab) \\ &\rhd_\beta (\lambda s.sab)\end{aligned}$$

Les fonctions de sélection (T pour tête et Q pour queue) vont alors devoir retourner respectivement leur premier argument (pour la tête) et leur second argument (pour la queue). On retrouve donc les booléens V et F :

$$T \equiv \lambda d.(dV)$$

$$Q \equiv \lambda d.(dF)$$

L'application de T à un doublet (construit avec D) nous donnera :

---

12 Il faut noter tout de même que les opérateurs Y (tels qu'ils sont décrits) ne pourront se coder que dans un langage fonctionnel non typé comme SCHEME (cf. chapitre 17), car ils ne possèdent pas de typage cohérent.

13 Prononcez « coudair »...

$$\begin{aligned}(T\ (D\ a\ b)) &\equiv (\lambda d.(dV))((\lambda t.\lambda q.\lambda s.stq)ab)\\ &\rhd_\beta ((\lambda t.\lambda q.\lambda s.stq)ab)V\\ &\rhd_\beta (\lambda s.sab)V\\ &\rhd_\beta (Vab)\\ &\rhd_\beta a\end{aligned}$$

Les doublets sont habituellement utilisés pour représenter des listes (via une liste chaînée de doublets). Il nous faut alors compléter notre jeu d'opérateurs avec la liste vide N et la fonction NULL (notée L) chargée de tester si son argument est la liste vide. Il est clair que L doit retourner F quand elle est appliquée à un doublet (un doublet ne représente pas la liste vide). On a donc :

$$\begin{aligned}(L\ (D\ a\ b)) &\rhd_\beta (L\ (\lambda s.sab))\\ &\equiv_\alpha F\end{aligned}$$

On en déduit que L est par exemple de la forme :

$$L \equiv \lambda d.d(\lambda t.\lambda q\ F)$$

Au contraire, on doit avoir *(L N)*$= V$ et donc :

$$N \equiv \lambda x.V$$

12.5.3 Les nombres entiers

Bien qu'il soit possible de représenter directement un entier n sous la forme d'une liste de longueur n (en utilisant les doublets de la section précédente), la représentation obtenue est un peu plus complexe que la représentation la plus usitée, dite des *nombres de Church*.

L'idée est de représenter un nombre n sous la forme d'une fonction qui appliquera n fois son premier argument à son second argument. Pour construire tous nos entiers, il nous suffit de définir le premier entier (zéro) et une fonction « successeur » qui permet d'obtenir l'entier suivant.

La représentation du nombre zéro va prendre en argument une fonction f et un argument x et appliquera f zéro fois à x. Il est donc égal à :

$$0 \equiv \lambda f.\lambda x.x$$

dont la ressemblance avec faux (F) est troublante. Le terme correspondant à la fonction successeur S prend en argument un nombre n (qui applique son premier argument (f) n fois à son second argument x) et retourne la représentation de son successeur *i.e.*, un terme qui appliquera son premier argument (f) $n+1$ fois à son second argument (x). On pourra donc appliquer f à x une première fois, puis appliquer f n fois au résultat obtenu en utilisant la représentation du nombre n :

$$S \equiv \lambda n.\lambda f.\lambda x.((nf)(fx))$$

On définit alors aisément les représentations des nombres :

$$(S\ 0) \equiv ((\lambda n.\lambda f.\lambda x.((nf)(fx)))(\lambda f.\lambda x.x))$$
$$\rhd_\beta \lambda f.\lambda x.fx$$

$2 \equiv \lambda f.\lambda x.(f(fx))$
$3 \equiv \lambda f.\lambda x.(f(f(fx)))$
etc.

Il est possible de comparer la représentation d'un nombre à 0 via le terme Z :

$$Z \equiv (\lambda n.\lambda x.\lambda y.n(Vy)x)$$

qui effectuera test et branchement : $(Znxy)$ se réduira à x si n est 0 et se réduira à y si n est la *représentation d'un nombre* non nul.

On peut construire de la même façon les fonctions d'addition, de multiplication, d'élévation à la puissance :

$$+ \equiv \lambda n_1.\lambda n_2.\lambda f.\lambda x.((n_1 f)((n_2 f)x))$$

et

$$\times \equiv \lambda n_1.\lambda n_2.\lambda f.\lambda x.((n_1(n_2 f))x)$$

$$\uparrow \equiv \lambda n_1.\lambda n_2.\lambda f.\lambda x.n_2 n_1 f x$$

Calculons par exemple la somme des termes 1 et de 0 :

$$(+\ 1\ 0) \equiv ((\lambda n_1.\lambda n_2.\lambda f.\lambda x.((n_1 f)((n_2 f)x))\lambda f.\lambda x.fx)\lambda f.\lambda x.x)$$
$$\rhd_\beta ((\lambda n_2.\lambda f.\lambda x.(((\lambda f.\lambda x.fx)f)((n_2 f)x)))\lambda f.\lambda x.x)$$
$$\rhd_\beta (\lambda f.\lambda x.(((\lambda f.\lambda x.fx)f)(((\lambda f.\lambda x.x)f)x)))$$
$$\rhd_\beta (\lambda f.\lambda x.((\lambda x.fx)x))$$
$$\rhd_\beta (\lambda f.\lambda x.fx)$$
$$\equiv_\beta 1$$

Il est par contre beaucoup plus délicat de définir un terme correspondant à la fonction « prédécesseur » (P) dans cette modélisation[14]. On peut définir[15] :

$$P = (\lambda n.(((n(\lambda a.\lambda b.b(S(aV))(aV)))(\lambda b.b00)))F))$$

Vérifions par exemple que le prédécesseur de 2 est bien 1 :

---

14 La représentation par les listes a l'avantage de fournir cet opérateur gratuitement, il suffit d'amputer une liste de son premier élément en utilisant le terme Q.

15 Cette expression assez complexe va en fait appliquer n fois une fonction qui la première fois ne fait rien (utilisation d'un drapeau), et qui les fois suivantes applique f. On arrive ainsi à construire $n-1$.

$(P2) \equiv ((\lambda n.(((n(\lambda a.\lambda b.b(S(aV))(aV)))(\lambda b.b00)))F))2)$
$\rhd_\beta (((2(\lambda a.\lambda b.b(S(aV))(aV)))(\lambda b.b00)))F))$
$\rhd_\beta (((\lambda z.(X(X)z)))(\lambda b.b00))F)$
ou $X \equiv (\lambda a.\lambda b.b(S(aV))(aV))$
$\rhd_\beta ((X(X(\lambda b.b00)))F)$
$\rhd_\beta ((X((\lambda a.\lambda b.b(S(aV))(aV))(\lambda b.b00)))F)$
$\rhd_\beta ((X(\lambda b.b(S((\lambda b.b00)V))((\lambda b.b00)V))F)$
On a $((\lambda b.b00)V) \rhd_\beta 0$ et donc
$\rhd_\beta ((X(\lambda b.b(S0)0))F)$
$\equiv (((\lambda a.\lambda b.b(S(aV))(aV))(\lambda b.b(S0)0))F)$
$\rhd_\beta ((\lambda b.b(S((\lambda b.b(S0)0)V))((\lambda b.b(S0)0)V))F)$
On a $((\lambda b.b(S0)0)V) \rhd_\beta (S0)$ et donc
$\rhd_\beta ((\lambda b.b(S(S0))(S0))F)$
$\rhd_\beta ((F(S(S0)))(S0))$
$\rhd_\beta (S0)$
$\equiv_\beta 1$

Il devient alors possible d'exprimer la fonction factorielle dans le cadre du λ-calcul pur, en utilisant les nombres de Church et la représentation des booléens :

$$\mathit{Fact} \equiv (Y(\lambda f.\lambda n.Zn1(\times n(f(Pn)))))$$

Soit, *in extenso*, en remplaçant les Y, Z, 1, $\times$, P par leur définition :

$$\begin{aligned}
\mathit{Fact} \equiv\ & ((\lambda x.((\lambda y.x(yy))(\lambda y.x(yy))))\\
& (\lambda f.\lambda n.(\lambda n.\lambda x.\lambda y.n((\lambda x.\lambda y.x)x)y)\\
& \quad n\\
& \quad (\lambda f.\lambda x.fx)\\
& \quad ((\lambda n_1.\lambda n_2.\lambda f.\lambda x.((n_1(n_2 f))x))\\
& \quad\ n\\
& \quad\ (f((\lambda n.(n(\lambda a.\lambda b.b\\
& \qquad\qquad ((\lambda n.\lambda f.\lambda x.((nf)(fx)))(a(\lambda x.\lambda y.x)))\\
& \qquad\qquad ((a(\lambda x.\lambda y.x)))(\lambda b.b(\lambda f.\lambda x.x)(\lambda f.\lambda x.x)))(\lambda x.\lambda y.y))n)))))
\end{aligned}$$

Cette expression est à comparer au code qu'il est nécessaire d'introduire dans une machine de Turing pour effectuer une simple incrémentation (cf. chapitre 4).

Il est clair que le λ-calcul pur, même si son usage est simplifié par l'utilisation d'abréviations (telles que les $Z, Y, 0, 1 \ldots$), n'a pas d'usage direct en tant que langage de programmation. Cependant, il peut servir de langage d'assemblage pour les langages fonctionnels (compilation en combinateurs).

Dans la majorité des cas, les langages de programmation fonctionnels s'appuient sur un λ-calcul appliqué, faisant intervenir des symboles de constantes ($+$, $\times \ldots$) ainsi que des règles de réduction spécifiques (dites de δ-réduction) qui permettent de donner leur sens à ces symboles.

Plutôt que d'utiliser, par exemple, les nombres de Church, dont l'utilisation est assez lourde, on préférera introduire les symboles de constantes $0, 1, 2 \ldots$ et $+, \times \ldots$ en y joignant des règles de δ-réduction donnant tout leur sens aux opérations d'addition, de multiplication, etc. Ces règles indiqueront, entre autres comment :

$(+\ 3\ 2) \rightsquigarrow 5$
$(\times\ 2\ 2) \rightsquigarrow 4$

ou plus généralement comment :

$(+n\ m) \rightsquigarrow n + m$
$(\times\ n\ m) \rightsquigarrow n.m$
etc.

Ces règles de réduction correspondent, dans un langage de programmation fonctionnel comme SCHEME, à l'appel de procédures externes permettant de traiter les entiers plus efficacement que les nombres de Church ne le permettent.

Le λ-calcul est aussi utilisé comme support en sémantique dénotationnelle, ou peut servir de langage de spécification. On utilisera alors un λ-calcul typé.

12.6 Termes du λ-calcul typé

Les termes du λ-calcul pur ont parfois une interprétation assez déroutante pour l'utilisateur habitué au concept de fonctions mathématiques, caractérisées par leur domaine et co-domaine. C'est par exemple le cas de l'abstraction $(\lambda x.(xx))$ dont la signification ensembliste n'est pas évidente.

Le λ-calcul typé permet de se rapprocher de la notion classique de fonctions en associant à chaque abstraction un domaine et un co-domaine. Ce gain en puissance d'expression ne va pas sans une contrepartie importante : la disparition de bon nombre de termes, non typables.

12.6.1 λ-calcul simplement typé

Dans ce système assez contraignant, les nombreux termes qui disparaissent sont des termes primordiaux (tels que les combinateurs de point fixe). Il s'agit historiquement de la première approche du typage en λ-calcul. Elle consiste essentiellement à limiter la formation des termes à des termes « naturellement » bien typés.

Il nous faut, avant de définir les termes du λ-calcul typé, définir ce qu'est un type :

Définition 12.15 – *Étant donné un ensemble de symboles appelés* types atomiques, *on définit un type inductivement par :*
- *tout type atomique est un type ;*
- *si α et β sont des types, alors $(\alpha \rightarrow \beta)$ est un type.*

Tout type résulte de l'application des règles précédentes un nombre fini de fois.

L'emploi de deux symboles α et β distincts dénotent *a priori* deux types distincts. Il est possible, afin d'augmenter la facilité d'expression, d'introduire de nouvelles règles de formation des types pour le produit cartésien (*si α et β sont des types, alors $(\alpha \times \beta)$ est un type*), l'union disjointe... Cela n'augmente pas la puissance d'expression des types puisque la notion de fonction seule permet déjà la création de paires (cf. section précédente sur les doublets, qui forment bien un produit cartésien).

Chaque type atomique dénote a priori un ensemble particulier. Il peut s'agir de l'ensemble des entiers naturels ($\mathbb{N}$), des booléens... Les types atomiques sélectionnés dépendront de l'utilisation.

Un type de la forme $(\alpha \rightarrow \beta)$ dénotera un ensemble donné de fonctions de domaine α et de co-domaine β. Tout comme pour l'abstraction du λ-calcul, on considère que $\rightarrow$ est associative à droite et l'on simplifie souvent :

$$(\alpha_1 \rightarrow (\alpha_2 \rightarrow \ldots \rightarrow (\alpha_n \rightarrow \beta) \ldots)) \text{ en } \alpha_1 \rightarrow \alpha_2 \rightarrow \ldots \rightarrow \alpha_n \rightarrow \beta$$

Exemples de types :

`num` $\rightarrow$ `num`

`num` $\rightarrow$ (`bool` $\rightarrow$ `num`)

Un terme du λ-calcul typé est, tout comme dans le λ-calcul pur, formé à partir de variables et de symboles de constantes, d'abstractions et d'applications. La différence essentielle est que chacun de ces termes se voit associer un type.

Pour chaque type α, on dispose d'un nombre infini dénombrable de variables du type α notées v^α et contenues dans un ensemble Σ^α auquel on peut rajouter un ensemble de symboles de constantes noté Γ^α. Nous dénoterons les types par des lettres minuscules grecques et continuerons à dénoter termes et variables de la même façon que précédemment, en ajoutant son type en exposant ($K^{\alpha \rightarrow \beta \rightarrow \alpha}$ dénotera donc un terme du λ-calcul de type $\alpha \rightarrow \beta \rightarrow \alpha$). On définit alors les termes du λ-calcul typé :

Définition 12.16 – *Un terme est obtenu en appliquant un nombre fini de fois les règles de formation suivantes :*

- **atome :** *Pour tout type α, toute variable v^α (élément de Σ^α) et tout symbole de constante k^α (élément de Γ^α) est un terme typé de type α ;*
- **application :** *Si $A^{\alpha \rightarrow \beta}$ et B^α sont deux termes typés de types respectifs $\alpha \rightarrow \beta$ et α, alors $(A^{\alpha \rightarrow \beta} B^\alpha)^\beta$ est un terme typé de type β ;*
- **abstraction :** *Si v^α est une variable de type α (élément de Σ^α) et si B^β est un terme typé de type β, alors $(\lambda v^\alpha . B^\beta)^{\alpha \rightarrow \beta}$ est un terme typé de type $\alpha \rightarrow \beta$.*

Chaque terme A^α (de type α) dénote un membre A de l'ensemble dénoté par le type α. Une abstraction $M^{\alpha \rightarrow \beta} \equiv (\lambda v^\alpha . B^\beta)^{\alpha \rightarrow \beta}$ dénote donc une fonction Υ de domaine α et de co-domaine β. L'application de cette abstraction $M^{\alpha \rightarrow \beta}$ à A^α aboutit à un terme $(M^{\alpha \rightarrow \beta} A^\alpha)^\beta$ qui dénote $\Upsilon(A)$ qui lui-même appartient à l'ensemble dénoté par le type β.

Bien souvent, on se permet de ne pas noter en exposant le type de tous les sous-termes intervenant dans un terme (en plus des simplifications habituelles). Ainsi, on écrira $(\lambda x^\alpha . \lambda y^\beta . x^\alpha)^{\alpha \rightarrow \beta \rightarrow \alpha}$ plutôt que l'expression complète :

$$(\lambda x^\alpha . (\lambda y^\beta . x^\alpha)^{(\beta \rightarrow \alpha)})^{(\alpha \rightarrow (\beta \rightarrow \alpha))}$$

L'écriture du terme doit tout de même être suffisamment complète pour que l'on puisse vérifier qu'elle est correctement typée (*i.e.*, qu'elle respecte les règles de formation d'un terme typé).

Exemples :

- $(\lambda x^\alpha . x^\alpha)^{\alpha \rightarrow \beta}$ est mal typé, en effet ce terme est construit à partir d'une règle d'abstraction impliquant une variable de type α et un terme de type α, il doit donc être de type $(\alpha \rightarrow \alpha)$.

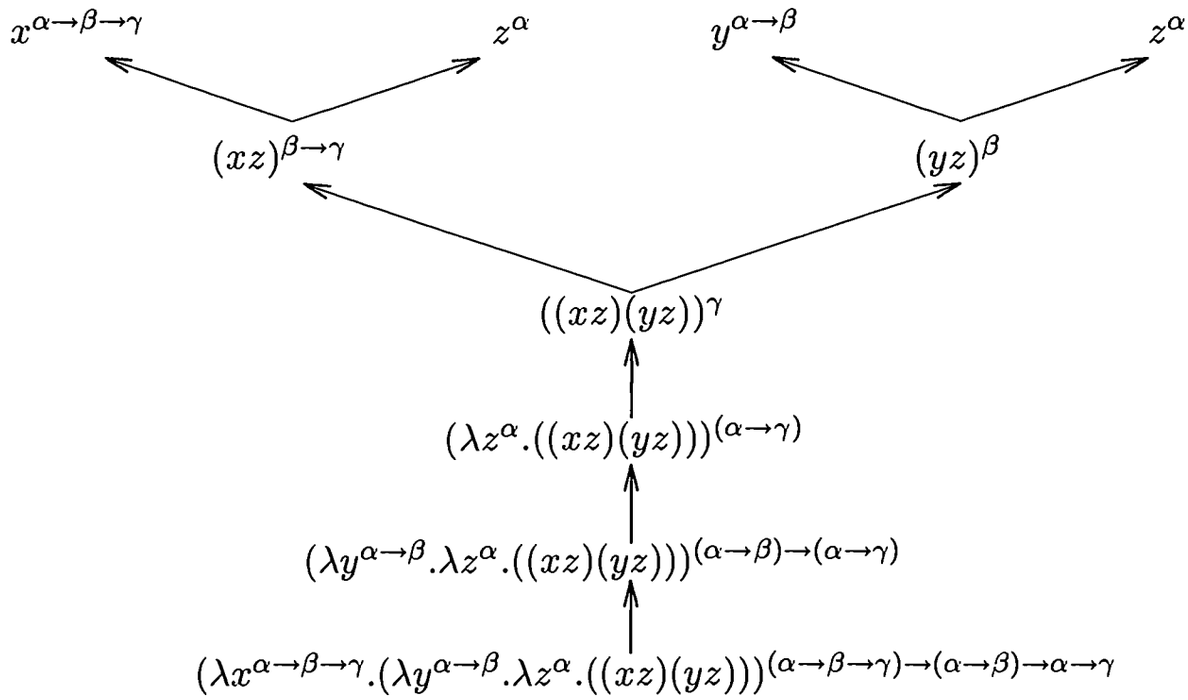


Figure 12.2 – Vérification du type de S

- $S = \lambda x^{\alpha\to\beta\to\gamma}.\lambda y^{\alpha\to\beta}.\lambda z^{\alpha}.((x^{\alpha\to\beta\to\gamma}z^{\alpha})(y^{\alpha\to\beta}z^{\alpha}))^{(\alpha\to\beta\to\gamma)\to(\alpha\to\beta)\to\alpha\to\gamma}$ est correctement typé. Cela peut se vérifier en examinant l'écriture complète du terme contenant les types de tous les sous-termes et en vérifiant qu'ils suivent les règles de formation des termes typés :
  $S = (\lambda x^{\alpha\to\beta\to\gamma}.$
  $(\lambda y^{\alpha\to\beta}.$
  $(\lambda z^{\alpha}.$
  $((x^{\alpha\to\beta\to\gamma}z^{\alpha})^{\beta\to\gamma}$
  $(y^{\alpha\to\beta}z^{\alpha})^{\beta})^{\gamma})^{\alpha\to\gamma})^{(\alpha\to\beta)\to(\alpha\to\gamma)})^{(\alpha\to\beta\to\gamma)\to(\alpha\to\beta)\to\alpha\to\gamma}$
- Cette « vérification de type » peut être aussi représentée sous la forme d'un arbre dont la racine sera le terme typé et dont chaque arc correspond à l'application d'une des règles de formation d'un λ-terme typé et dont les feuilles seront les variables typées. La figure 12.2 montre l'arbre de vérification du typage du terme S précédent.

Les principales définitions du λ-calcul pur se retrouvent directement dans le λ-calcul typé (substitution, β-redex, β-réduction...). L'unique différence est dans la substitution, qui ne permet de substituer à une variable de type α donné qu'un terme de même type.

D'importantes différences apparaissent par contre dans les théorèmes qui en découlent. Le théorème de Church-Rosser reste valide. Le théorème de standardisation se voit fortement renforcé :

Théorème 12.7 – *Tous les termes du λ-calcul typé ont une β-forme normale. Cette forme est atteinte indépendamment de la stratégie de réduction.*

En terme de systèmes de réécritures, on dit que le système obtenu est *nœthérien*. Tout comme la propriété de confluence du λ-calcul non typé, cette propriété peut disparaître lors de l'introduction de règles de δ-réduction. En particulier l'ajout d'une règle de δ-

réduction indiquant la commutativité d'une opération (Ex : $x + y \rhd_\delta y + x$) rend le système non-nœthérien.

On déduit de cette nœtherianité une conséquence importante :

Théorème 12.8 – *La β-équivalence est décidable en λ-calcul typé.*

Il suffit en effet de ramener les termes à leur forme normale et de les comparer syntaxiquement pour décider de leur β-équivalence.

Une seconde conséquence importante est que tous les termes du λ-calcul pur qui n'avaient pas de forme normale disparaissent du paysage, en particulier l'abstraction $(\lambda x.xx)$. Avec eux disparaissent les combinateurs de point fixe qui ne peuvent pas être construits suivant les règles de formation des λ-termes typés.

Il est donc impossible de faire de la récursivité en λ-calcul typé sans constante. Il est alors nécessaire de rajouter des symboles de constantes et des règles de δ-réduction spécifiques. On perd ainsi la propriété de décidabilité.

Ce système de typage se rapproche des langages fortement typés classiques où chaque variable est typée à sa déclaration et où le compilateur se charge de vérifier la cohérence entre les types des fonctions utilisées et les arguments reçus. Ce type de système est assez contraignant (il demande un travail de typage important de la part du programmeur) et entraîne une réutilisabilité faible (il est nécessaire d'écrire une nouvelle fonction identité pour chaque nouveau type traité).

12.6.2 Affectation de type en λ-calcul non typé

L'introduction de type dans les termes du λ-calcul a pour but (hormis la disparition de termes étranges tels que $(\lambda x.(xx))$) le rapprochement avec la notion de fonctions mathématiques classiques. Pour certains termes, cela ne se fait pas sans une certaine lourdeur d'utilisation. Si l'on considère par exemple le terme non typé $I \equiv \lambda x.x$, il est clair que sa définition précède toute information de domaine ou de co-domaine alors que le système de typage précédent va entraîner la définition d'une infinité d'abstractions telles que :

$$I^\alpha \equiv \lambda x^\alpha.x^\alpha$$

pour chaque type α (on dit que le terme I est polymorphe). Il semblerait plus naturel, et plus simple, de partir de termes non typés, qui contiennent l'essence de l'algorithme, en leur adjoignant ensuite un type. Le problème est alors d'affecter un type à un terme du λ-calcul non typé.

Types avec variables

Afin de pouvoir traiter sans lourdeur des fonctions ayant plusieurs types possibles (telles que l'identité) il est nécessaire d'introduire des variables dans nos types. Nous allons donc faire quelque peu évoluer notre définition des types :

Définition 12.17 – *Étant donnés un ensemble de symboles appelés* constantes de types *ou* types atomiques *et un ensemble de variables de types Δ, on définit un type inductivement par :*

- *tout type atomique, toute variable de type (élément de Δ) est un type ;*
- *si α et β sont des types, alors $(\alpha \rightarrow \beta)$ est un type.*

Tout type est obtenu par application des règles précédentes un nombre fini de fois.

De façon traditionnelle, les variables de types seront notées « $a, b, c, d, e \ldots$ », les minuscules « $u, v, w \ldots$ » étant réservées aux variables des λ-termes. Les lettres minuscules grecques « $\alpha, \beta \ldots$ » dénoteront des types (avec variables). Un type contenant une variable sera dit polymorphe, un type sans variable sera dit monomorphe.

Il est possible de substituer un type α à une variable de type a dans un type polymorphe β. Nous noterons le type ainsi obtenu $[\alpha/a]\beta$, qui est alors une *instance* du type β. Les variables intervenant dans un type sont implicitement quantifiées universellement (un type polymorphe représente l'ensemble de ses instances possibles).

Exemple : $\alpha \equiv a \rightarrow b$ est un type. Il est possible de substituer le type $\beta \equiv c \rightarrow c$ à b dans α. On obtient ainsi $[\beta/b]\alpha \equiv a \rightarrow (c \rightarrow c)$. Le type $a \rightarrow b$ représente en fait l'ensemble des types de la forme $a \rightarrow b$ où a et b sont deux types quelconques.

On s'intéressera par la suite à l'affectation de type à des termes du λ-calcul pur. Étant donnés un terme X et un type α, on notera $X \in \alpha$ le fait que le type α soit affecté à X.

Système de typage

Afin de pouvoir affecter un type α à un terme X quelconque, il va être nécessaire de connaître (ou de supposer) le type de ses composants élémentaires : les variables. Ainsi, le terme $\lambda x.x$ doit se voir affecter le type $(a \rightarrow b) \rightarrow (a \rightarrow b)$ si l'on suppose que x est de type $a \rightarrow b$. On va donc faire des hypothèses sur le type des variables.

Définition 12.18 – *Nous appellerons* contexte de typage *un ensemble d'affectations de types à des variables de la forme* $\{x \in \alpha, y \in \beta \ldots\}$ *où* x, y *sont des variables de* λ*-terme toutes distinctes et* α, β *des types.*

Le système de typage[16] est formé d'un ensemble de règles d'inférence. Les formules manipulées par le système d'inférence sont des formules de la forme $C \vdash X \in \alpha$ exprimant que dans le contexte de typage C, le terme X se voit affecter le type α. Lorsqu'un contexte de typage est vide, on écrit $\vdash X \in \alpha$ et la formule $X \in \alpha$ est un théorème du système.

Règle d'inférence 12.1 – *(A)* $\dfrac{}{x_1 \in \alpha_1 \ldots, x_n \in \alpha_n \vdash x_i \in \alpha_i}$

---

16 Un système de typage équivalent peut être construit dans le cadre de la logique combinatoire. Il est formé de deux schémas d'axiome donnant les types de K et S :

$$K \in \alpha \rightarrow (\beta \rightarrow \alpha)$$

$$S \in ((\alpha \rightarrow (\beta \rightarrow \gamma)) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma)$$

et d'une règle d'inférence associée à l'application :

$$\frac{U \in (\alpha \rightarrow \beta)\ V \in \alpha}{C \vdash (UV) \in \beta}$$

Curry a noté (isomorphisme de Curry/Howard) que si l'on interprète la flèche « $\rightarrow$ » comme une implication, cette règle d'inférence est l'équivalent du *« modus-ponens »* et les deux schémas d'axiome sont les schémas d'axiome de la logique propositionnelle intuitionniste. Un type peut alors être interprété comme une formule du calcul propositionnel, l'affectation de ce type à un terme comme la preuve de cette formule.

Informellement : « Si le contexte de typage C indique que le type α_i est affecté à x_i, alors il est possible de l'inférer »...

Règle d'inférence 12.2 – *(B)* $\dfrac{C \vdash U \in (\alpha \to \beta)\ C \vdash V \in \alpha}{C \vdash (UV) \in \beta}$

Informellement : « Si le contexte de typage permet d'inférer que U a le type $(\alpha \to \beta)$ et que V a le type α, alors on peut inférer que (UV) a le type β ».

Règle d'inférence 12.3 – *(C)* $\dfrac{\{x \in \alpha\} \cup C \vdash U \in \beta}{C \vdash (\lambda x.U) \in (\alpha \to \beta)}$

Informellement : « Si, dans le contexte de typage C, on peut, en supposant que x a le type α, inférer que U a le schéma de type β, alors on peut inférer que $(\lambda x.U)$ a le type $(\alpha \to \beta)$ ». Dans cette règle, l'hypothèse « $x \in \alpha$ » apparaît dans le contexte de typage de la prémisse mais a disparu de la conclusion. On dit qu'elle a été déchargée[17].

Montrons que :

$$S \equiv (\lambda u.\lambda v.\lambda w.((uw)(vw)))$$

admet pour type

$$(\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$$

On a d'après (A) :
- $u \in \alpha \to \beta \to \gamma, v \in \alpha \to \beta, w \in \alpha \vdash v \in \alpha \to \beta$;
- $u \in \alpha \to \beta \to \gamma, v \in \alpha \to \beta, w \in \alpha \vdash w \in \alpha$;
- donc d'après (B) : $u \in \alpha \to \beta \to \gamma, v \in \alpha \to \beta, w \in \alpha \vdash (vw) \in \beta$.
- En procédant de même avec u et v on obtient :

$$u \in \alpha \to \beta \to \gamma, v \in \alpha \to \beta, w \in \alpha \vdash (uv) \in \beta \to \gamma$$

- On a donc par application de (B), $u \in \alpha \to \beta \to \gamma, v \in \alpha \to \beta, w \in \alpha \vdash ((uw)(vw)) \in \gamma$.
- Il suffit maintenant d'appliquer (C) une fois pour chacune des variables sans oublier de décharger les hypothèses correspondantes. On a donc :

$$u \in \alpha \to \beta \to \gamma, v \in \alpha \to \beta, [w \in \alpha] \vdash (\lambda w.((uw)(vw))) \in \alpha \to \gamma$$

- puis $u \in \alpha \to \beta \to \gamma, [v \in \alpha \to \beta] \vdash (\lambda v.\lambda w.((uw)(vw))) \in (\alpha \to \beta) \to \alpha \to \gamma$;

---

17 Pour les deux règles précédentes, le contexte de typage passe des prémisses aux conclusions sans modification, et il est donc rarement mentionné dans la pratique. On écrira par exemple :

$$\frac{U \in (\alpha \to \beta)\ V \in \alpha}{(UV) \in \beta}$$

sans préciser le contexte de typage (qui sera implicitement celui utilisé pour démontrer $U \in (\alpha \to \beta)$ et $V \in \alpha$). Lorsque la règle d'inférence (C) est appliquée, on devra donc spécifier que le contexte de typage ne passe pas inchangé en indiquant qu'une des hypothèses de typage disparaît du contexte. L'attitude la plus courante consiste à mettre l'hypothèse déchargée entre crochets ($[x \in \alpha]$) ou tout simplement à la barrer.

– puis enfin :

$$[u \in \alpha \to \beta \to \gamma] \vdash (\lambda u.\lambda v.\lambda w.((uw)(vw))) \in (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$$

Le contexte final est vide, on a donc démontré que :

$$\vdash (\lambda u.\lambda v.\lambda w.((uw)(vw))) \in (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$$

Le type d'un terme X n'est pas unique. On a vu que pour tout type α, on a $\vdash \lambda x.x \in \alpha \to \alpha$. On a donc $\vdash \lambda x.x \in a \to a$, pour $\alpha = a$, mais aussi $\vdash \lambda x.x \in (a \to b) \to (a \to b)$ (par exemple).

Théorème 12.9 – *On peut, pour chaque terme X admettant un type, définir un type principal, unique aux renommages de variables de type près, dont tout type de X est une instance obtenue en substituant un type à une ou plusieurs variables du type principal.*

Le type principal correspond à un type minimal dont tout type peut se déduire par instanciation. (Ainsi, le type principal de $\lambda x.x$ est $a \to a$ (aux renommages de variables près), une instanciation possible est $(\mathtt{num} \to \mathtt{num})$).

Il faut remarquer que tout terme du λ-calcul pur n'admet pas forcément un type.

Définition 12.19 – *Un terme du λ-calcul pur X tel que ($\exists \alpha$ tel que $\vdash X \in \alpha$) est dit stratifié.*

L'ensemble des résultats obtenus pour les termes du λ-calcul typé se retrouve sur les termes du λ-calcul pur stratifiés. En particulier, la β-équivalence de deux termes stratifiés est décidable. Naturellement, les termes tels que $\lambda x.(xx)$ ne sont pas stratifiés.

Notons que grâce à la présence de variables dans nos types, le système permet *l'expression* de fonctions polymorphes mais ne permet pas leur *utilisation* polymorphe (aucune règle d'inférence ne permet la substitution d'un type à une variable de type). Considérons une fonction polymorphe par excellence : $I \equiv (\lambda x.x)$ et considérons l'abstraction $G \equiv (\lambda y.(yy))$ appliquée à I.

$$(GI) \equiv (\lambda y.(yy))(\lambda x.x)$$

Bien que G n'ait intuitivement pas de type dans le cas général, il est clair que lorsque G est appliqué à l'identité, l'expression obtenue a un sens et son type est intuitivement le même que celui de l'identité (à laquelle (GI) est β-équivalent) *i.e.*, $a \to a$. Cependant, si nous essayons de typer cette expression au moyen des règles d'inférence qui précèdent, nous n'aboutissons à rien. Il est en effet nécessaire de faire une supposition sur le type de y ; que l'on suppose que $y \in (a \to a)$ ou que $y \in (a \to a) \to (a \to a)$, il est impossible d'inférer un type pour l'expression (yy).

Ce système de typage est donc trop dur puisqu'il ne permet pas de typer des expressions intuitivement correctes. Le problème provient en fait des abstractions. Ici, dans $(\lambda y.(yy))$, bien que y ait des possibilités polymorphiques (la variable y est liée à l'identité qui est polymorphe), ces possibilités ne peuvent pas être exploitées à l'intérieur de l'abstraction $(\lambda y.(yy))$ car un seul type doit être affecté à y, et seul ce type peut être utilisé par la suite pour typer les expressions contenues dans l'abstraction (ici (yy)). Il faudrait pouvoir donner à y, une première fois (première occurrence de y) le type $(b \to b) \to (b \to b)$, et une deuxième fois (seconde occurrence) le type $(b \to b)$, tous deux instances du type de $I : a \to a$.

12.6.3 Un système de typage polymorphique

Afin de pouvoir utiliser une fonction visiblement polymorphe avec des types distincts, deux ajouts sont nécessaires. Il va falloir introduire une nouvelle forme dans notre langage des termes du λ-calcul et affiner notre système de typage (Damas and Milner 1982).

Extension du λ-calcul

Comme on a pu s'en rendre compte, la raison principale de la limitation du système de typage précédent est que les variables paramètres formels d'une abstraction ne peuvent avoir qu'un type, une fois cette abstraction appliquée. L'abstraction $G \equiv (\lambda y.(yy))$ n'a pas de sens (et de type valide) prise isolément, mais prend un sens dès lors qu'elle est appliquée à une fonction suffisamment polymorphe (telle que l'identité).

Il est donc possible d'améliorer notre système de typage en traitant de façon spécifique les cas où une abstraction est appliquée à un argument. Dans notre cas, il est visible que le β-redex (G 1+), où 1+ est le symbole dénotant la fonction successeur de type num→num n'est pas typable (appliquer une fonction de type num→num à elle-même n'a pas de sens) de même qu'il est visible que $(G\ (\lambda x.0))$ en a un ($(\lambda x.0)$ est le terme constant partout égal à 0, en particulier lorsqu'il est appliqué à lui-même).

Il faut donc introduire une nouvelle règle d'inférence spécifique (D) pour le typage des β-redex. L'inconvénient majeur de cette approche est qu'elle débouche sur un système de règles d'inférence qui est ambigu puisqu'il va être possible d'appliquer aux β-redex la nouvelle règle (D) (qui reste à établir) ou la règle (B) de typage des applications alors que seule (D) devrait être appliquée.

Plutôt que d'introduire une stratégie d'application des règles d'inférence, on introduit une nouvelle forme syntaxique, presque équivalente à la forme $((\lambda x.U)V)$ et qui permettra le polymorphisme. Il s'agit de la forme `let`. Du point de vue de la β-réduction, la forme `let` $x = V$ `in` U est *strictement équivalente* à la forme $((\lambda x.U)V)$. Elle ne modifie donc en rien les propriétés déjà obtenues dans le cadre du λ-calcul.

La différence essentielle intervient dans le système de typage, avec l'introduction d'une règle de typage spécifique qui entraîne une distinction entre les variables liées dans une abstraction, qui ne permettront pas une application polymorphe (ces variables sont dites *non-génériques*), et les variables *génériques* liées dans un `let`, qui vont permettre l'application polymorphe. Cette distinction nécessite l'introduction d'un raffinement du système des types.

Schémas de type

Le but de notre système de typage reste l'affectation de types à des termes du λ-calcul. Cependant, il va être nécessaire (pendant l'inférence de type) de distinguer les variables génériques (liées dans un `let`) qui pourront être utilisées de façon plus souple que les variables non-génériques. C'est le rôle des schémas de type.

Définition 12.20 – *Un schéma de type est formé d'un type α et d'un sous-ensemble (éventuellement vide) $\mathcal{V} = \{a_1, \ldots, a_n\}$ des variables de type contenues dans α. On le note habituellement $\forall a_1 \ldots, \forall a_n\ :\ \alpha$, parfois simplifié en $\forall a_1, \ldots, a_n\ :\ \alpha$.*

Nous dénoterons les schémas de type par des lettres minuscules grecques surlignées ($\overline{\alpha}\ldots$).

Les variables quantifiées dans un schéma de type sont liées par le quantificateur. Les autres sont dites libres. L'utilisation des schémas sera la suivante :

- un schéma de type avec des variables quantifiées sera le type d'une variable générique ;
- un type (ou schéma de type sans variable quantifiée) sera associé à une variable non générique.

Tout comme pour un type, il est possible de substituer un type α à une variable de type libre a dans un schéma de type $\overline{\beta}$ (cela peut nécessiter le renommage de certaines variables liées s'il y a collision de noms). Nous noterons le schéma de type ainsi obtenu $[\alpha/a]\overline{\beta}$, qui est alors une *instance* du schéma de type $\overline{\beta}$.

Exemple : Le schéma de type $\overline{\alpha} \equiv \forall a \ : a \rightarrow b$ peut s'instancier :

- en substituant le type $c \rightarrow c$ à b, on obtient alors :

$$[c \rightarrow c/b]\overline{\alpha} \equiv \forall a \ : a \rightarrow (c \rightarrow c)$$

- en substituant le type $a \rightarrow a$ à b, il va alors être nécessaire de renommer la variable liée a de $\overline{\alpha}$. On obtient par exemple le schéma de type :

$$[a \rightarrow a/b]\overline{\alpha} \equiv \forall d \ : d \rightarrow (a \rightarrow a)$$

Il est, de plus, possible d'appliquer une substitution (éventuellement vide) σ sur une partie des variables liées $b_1 \ldots, b_n$ d'un schéma de type $\overline{\beta} = \forall b_1 \ldots, b_n \ : \beta$. Le schéma de type obtenu est alors $\overline{\beta'} = \forall b_i \ : \sigma(\beta)$ où les b_i forment le sous-ensemble de $\{b_1 \ldots, b_n\}$ des variables n'ayant pas subi de substitution. Il est appelé « instance générique » de $\overline{\beta}$, noté $\overline{\beta'} \preccurlyeq \overline{\beta}$.

Exemple : on peut substituer dans le schéma de type précédent ($\overline{\alpha}$) $c \rightarrow c$ à a. On obtient une instance générique ($\overline{\alpha'} \preccurlyeq \overline{\alpha}$) de $\overline{\alpha}$:

$$\overline{\alpha'} \equiv (c \rightarrow c) \rightarrow b$$

Une affectation de type devient maintenant une affectation de schéma de type, de la forme $X \in \forall a_1, \ldots, a_n \ : \alpha$ signifiant que X se voit affecter le schéma de type $\forall a_1, \ldots, a_n \ : \alpha$.

Un contexte de typage devient alors un ensemble d'affectations de schémas de type à des variables (ou symboles de constantes en λ-calcul appliqué) de terme.

Nous noterons C_x le contexte de typage obtenu à partir du contexte C en supprimant toute hypothèse de typage sur x. On notera $\mathcal{G}_C(\beta)$ le schéma de type défini par le type β, et dont les variables quantifiées sont les variables de β qui n'apparaissent pas libres dans le contexte de typage C.

Ce contexte de typage $\mathcal{G}_C(\beta)$ donne le statut générique à toutes les variables dont le type contient des variables de type libres dans le contexte. Si une variable est liée par une abstraction, il a été nécessaire de faire une hypothèse sur son type, et ses variables de type apparaissent alors libres dans le contexte. Elle ne peut donc être rendue générique. Si elle a été liée par un `let` (voir règle (D')), elle est rendue générique (si elle n'a pas été, par ailleurs, liée par une abstraction).

Règle d'inférence 12.4 – *(A')* $\dfrac{}{x_1 \in \overline{\alpha_1} \ldots, x_n \in \overline{\alpha_n} \vdash x_i \in \beta_i}$ *où* $\beta_i \preccurlyeq \overline{\alpha_i}$

Informellement, si un contexte de typage indique que x_i a un schéma de type $\overline{\alpha_i}$, alors on peut inférer que x_i a le type β_i instance générique de $\overline{\alpha_i}$. Cette règle va permettre l'utilisation d'une variable générique (dont le type contient des variables liées) avec plusieurs types distincts (qui sont tous des instances génériques du type de la variable).

Règle d'inférence 12.5 – *(B')* $$\frac{C \vdash U \in \beta \to \gamma \; C \vdash V \in \beta}{C \vdash (UV) \in \gamma}$$

Informellement, si l'on peut inférer que U est de type $\beta \to \gamma$ et que V est de type β, alors (UV) est de type γ (inchangée).

Règle d'inférence 12.6 – *(C')* $$\frac{\{x \in \beta\} \cup C \vdash U \in \gamma}{C \vdash (\lambda x.U) \in \beta \to \gamma}$$

Signification inchangée.

Règle d'inférence 12.7 – *(D')* $$\frac{C \vdash V \in \beta \; \{x \in \mathcal{G}_C(\beta)\} \cup C_x \vdash U \in \gamma}{C \vdash (\texttt{let}\ x = V\ \texttt{in}\ U) \in \gamma}$$

C'est cette dernière règle qui rend les variables génériques. Toutes les variables dont le type contient des variables de type n'apparaissant pas libres dans le contexte vont être rendues génériques. Les variables qui ne sont pas rendues génériques sont celles qui ont déjà été liées dans une abstraction et sur lesquelles il a fallu faire une hypothèse de typage.

Considérons à nouveau nos abstractions G et I dans l'application de G à I :

$$(GI) \equiv ((\lambda y.(yy))(\lambda x.x))$$

Cette expression peut s'écrire sans modification pour la β-réduction :

$$\texttt{let}\ y = (\lambda x.x)\ \texttt{in}\ (yy)$$

On sait que la règle (B), inchangée par rapport au système précédent permet d'inférer que le type de I est $(a \to a)$. Pour appliquer la règle (D'), il nous faut d'une part construire le contexte de typage $C' = \{y \in \mathcal{G}_C(a \to a)\} \cup C_y$:

$$C' = \{y \in \forall a \ : a \to a\}$$

puisque a est une variable de type libre dans C, et d'autre part inférer le type de (yy) dans ce contexte.

Plaçons-nous dans ce contexte C'. La règle (A') permet d'associer à la première occurrence de y le type $(b \to b) \to (b \to b)$ (car c'est une instance générique du schéma de type $\forall a \ : a \to a$) et d'associer à la seconde occurrence de y le type $(b \to b)$ (pour la même raison). On infère alors (par (B')) que $C' \vdash (yy) \in (b \to b)$.

Si l'on revient maintenant à la règle (D') dont les deux prémisses ont été remplies, il vient que :

$$C \vdash \texttt{let}\ y = (\lambda x.x)\ \texttt{in}\ (yy) \in (b \to b)$$

Ce nouveau système de typage, plus complexe, permet donc l'inférence de type sur des fonction polymorphes utilisées de façon polymorphe. C'est le système de typage actuellement utilisé dans la majorité des langages fonctionnels pratiquant l'inférence de type. Il permet d'obtenir la sécurité des langages fortement typés, mais aussi une certaine légèreté d'utilisation (inutile de déclarer le type de tous les paramètres) et surtout une réutilisabilité accrue des fonctions (une fonction polymorphe peut être réutilisée directement sur tous les types). On retrouve ainsi certaines des possibilités du langage ADA (packages génériques).

Partie III

Techniques de l'Intelligence Artificielle

AI is in the eye of the beholder.
To academics, it is "Anything Impossible,"
To the military, it is "Anything Invincible,"
To the marketing departments of software firms,
it is "Anything Improved."

— *Anonyme*

CHAPITRE 13

Méthodes faibles

Jean-Marc Alliot

13.1 Introduction

Pour pouvoir résoudre un problème, quel qu'il soit, il faut avant tout le poser correctement, l'analyser, et définir les moyens pour le résoudre. Le but de ce chapitre est de développer les méthodes permettant de réaliser systématiquement ces trois opérations.

Il portera donc sur cinq points :
- la définition de l'espace d'états du problème ;
- la définition d'un système de production permettant de se déplacer à l'intérieur de l'espace d'états ;
- la définition des structures de représentation ;
- la définition de stratégies de contrôle efficaces ;
- l'analyse des caractéristiques du problème permettant de choisir une représentation et une stratégie de contrôle adaptées.

Les méthodes pour la résolution (et la représentation) que nous décrivons dans ce chapitre sont appelées *méthodes faibles*. Elles sont basées pour la plupart sur des techniques de parcours de graphe et peuvent être adaptées à de nombreux domaines. Leur efficacité est généralement mauvaise pour un problème particulier, car elles sont incapables de maîtriser l'explosion combinatoire très importante que ce type de représentation induit (cette analyse a conduit à la réalisation de *« systèmes experts »* dont nous aurons l'occasion de reparler). Comme a pu le montrer la théorie de la complexité (cf. chapitre 11), certains problèmes semblent combinatoires de façon inhérente, et ce n'est donc pas surprenant.

13.2 Espaces d'états

Nous posons dans ce paragraphe l'ensemble des postulats de base de la résolution de problèmes.

13.2.1 Principes généraux

Définition 13.1 – Variables d'état – *Tout problème est relatif à un certain ensemble d'objets que l'on peut décrire sous forme de* variables d'état *du problème.*

Définition 13.2 – État d'un problème – *Un état du problème est l'ensemble des valeurs que prennent ses variables d'état à un instant donné.*

Définition 13.3 – Espace d'états – *L'espace d'états d'un problème est l'ensemble des états possibles pour le problème considéré.*

On parle plus volontiers d'espace d'états que d'ensemble d'états. Nous verrons tout à l'heure que ce choix terminologique est justifié : *ensemble* évoque une collection d'objets sans rapport les uns avec les autres, alors qu'*espace* implique déjà des notions de relation entre les objets (déplacements dans l'espace). Notons tout de même une légère ambiguïté : qu'est-ce qu'un état *possible* ? En fait, nous verrons que les états possibles sont les états qui peuvent être générés à partir des états initiaux en suivant les règles de production du système. Mais même ainsi, il n'est pas toujours simple de savoir si un état donné est possible ou non (aux échecs cela s'appelle *réaliser une analyse rétrograde*) et cela peut même se révéler impossible[1]. D'autre part, il est des problèmes où il est nécessaire d'étendre cette notion d'états possibles pour rendre le problème plus facile à résoudre. Nous y reviendrons dans les exemples développés dans ce chapitre.

Les définitions que nous venons de présenter marquent bien les limites des mécanismes que nous allons décrire dans cette partie. Les problèmes que nous allons examiner doivent être parfaitement connus et déterminés, puisque nous devons pouvoir ramener leur description à un ensemble fini de variables d'état. D'autre part, il doit être possible de discrétiser le problème d'une façon ou d'une autre, afin d'avoir un espace d'états au plus dénombrable, pour qu'il soit descriptible par une machine de Turing.

13.2.2 Exemples

Nous allons présenter quelques exemples de problèmes classiques que l'on sait facilement décrire sous forme d'espace d'états :

Les missionnaires et les cannibales : Ce problème, très classique, est connu sous bien d'autres formes (problèmes des lions et des dompteurs, ou du lama, de la chèvre, du lion, et du berger). Il remonte au Moyen Âge. Voici un des énoncés :

> « Trois cannibales et trois missionnaires doivent traverser un fleuve en empruntant une barque qui ne peut contenir que deux personnes. Or, si sur une des deux rives, à un moment quelconque, on trouve plus (strictement) de cannibales que de missionnaires, les premiers nommés font chauffer la marmite. Les missionnaires doivent donc trouver une méthode pour qu'à aucun moment ils ne soient en infériorité numérique. »

Un état peut être représenté par un triplet (X, Y, P) avec $0 \leq X \leq 3, 0 \leq Y \leq 3$ et $P = D$ ou $P = G$. X représente le nombre de missionnaires sur la rive gauche, Y le nombre de cannibales sur cette même rive et P représente la position de la barque (D pour rive droite et G pour rive gauche). Il est clair que nous avons ainsi défini l'espace d'état de notre problème. L'état initial est $(3, 3, G)$ et on cherche à atteindre l'état final $(0, 0, D)$.

---

1 On peut utiliser le problème de Post pour le montrer.

La démonstration de théorèmes : Considérons le problème de construction d'un système réalisant la démonstration automatique de théorème en arithmétique formelle. L'espace d'états est évidemment l'ensemble des formules bien formées.

Le jeu d'échecs : Le jeu d'échecs peut aisément être ramené à l'état de chacune des 64 cases à un instant donné. Le nombre de variables d'états (64) est fini, les valeurs que peuvent prendre ces variables également : une case peut contenir un pion, une tour, un cavalier, un fou, une dame ou un roi de l'une ou l'autre couleur, ou peut être vide, soit 13 valeurs différentes possibles. D'autre part le problème est parfaitement discret : on passe d'un état à un autre par une modification discrète d'une ou plusieurs variables.

Quels sont les états possibles du système ? Nous pourrions prendre en première approximation l'ensemble des affectations possibles de chacune des 13 valeurs dans les 64 variables. Mais il est clair que tous ces états ne sont pas possibles. En particulier, on ne peut jamais trouver deux rois de la même couleur simultanément sur l'échiquier, ni de pion blanc sur la première rangée, ni de pion noir sur la huitième rangée. Nous pouvons donc exclure ces états de notre espace. Mais il est d'autres états qui sont impossibles, quoi que plus difficiles à déterminer. Faut-il en tenir compte ? Nous nous trouvons là face au dilemme que nous soulignions tout à l'heure : déterminer les états possibles d'un système de façon précise peut se révéler extrêmement difficile. On se contente souvent de considérer comme espace d'états un sur-ensemble de l'espace des états possibles, plus facile à définir.

Le problème du voyageur de commerce : Nous avons déjà évoqué ce problème dans le chapitre 11 (théorie de la complexité). Il s'agit de trouver le plus court circuit passant par n villes sans jamais passer deux fois dans la même ville. L'espace d'états du problème est alors un ensemble composé de tous les singletons, doublets, triplets, n-uplets ne contenant qu'une seule fois une ville. Ainsi, pour un problème à quatre villes, le doublet $(1, 3)$ signifierait que le voyageur a commencé son circuit par la ville 1, puis qu'il est allé dans la ville 3 et qu'il lui reste deux villes à parcourir. Seuls les quadruplets sont susceptibles d'être des solutions du problème. L'espace d'états que nous avons défini est plus vaste que l'espace des solutions possibles du problème.

Le jeu de poker sans écart : L'espace[2] d'états est constitué de l'ensemble des mains possibles pour le joueur représenté par le programme, augmenté de la séquence de toutes les enchères possibles. Remarquons que nous ne considérons pas l'ensemble des mains de l'adversaire : c'est une caractéristique des jeux à cartes cachées sur laquelle nous reviendrons.

Pour mieux comprendre l'intérêt d'espace d'états plus vaste que l'espace contenant les solutions possibles, il nous faut aborder la notion de *marche vers la solution* et de *système de production.*

---

2 Nous nous contenterons du jeu de Poker sans écart, qui est plus simple à présenter que le jeu de Poker traditionnel, mais l'ensemble des résultats que nous énoncerons peuvent aisément se généraliser.

13.3 Systèmes de production

13.3.1 Raisonnement en chaînage avant

Résoudre un problème par un raisonnement déductif, consiste à identifier l'*état initial*, l'*état final* et un mécanisme permettant de décrire les règles régissant le passage d'un état à un autre : le *système de production*. Les notions d'état initial et d'état final sont claires, en revanche il est bon de préciser celle de système de production.

Définition 13.4 – Système de production – *Un système de production est constitué d'un ensemble de règles, les règles de production, qui permettent à partir d'un état du problème donné, de générer l'ensemble des états que l'on peut légalement atteindre depuis cet état particulier.*

Définition 13.5 – Solution déductive d'un problème – *Obtenir une solution déductive d'un problème consiste à passer de l'état initial à un état final*[3] *en suivant une chaîne d'états intermédiaires, chaque état intermédiaire étant généré en accord avec le système de production.*

Ce type de système de production est dit à chaînage avant.

Cette notion de règle de production n'est pas si éloignée des systèmes de production tels que nous les avons définis en théorie des langages. Comme nous allons le voir, la définition des règles de production n'est pas toujours chose facile. On peut généralement définir plusieurs systèmes de production possibles, souvent très différents.

Nous avons d'autre part affirmé que les notions d'état initial et d'état final étaient claires. Il n'est cependant pas toujours évident de reconnaître un état comme étant final. Ainsi, aux échecs, vérifier que l'adversaire est mat n'est pas une chose triviale à formaliser.

13.3.2 Raisonnement en chaînage arrière

Ce type de raisonnement est aussi appelé raisonnement inductif. La seule différence entre raisonnement inductif et raisonnement déductif se trouve dans la définition de la solution :

Définition 13.6 – Solution inductive d'un problème – *Obtenir une solution inductive d'un problème consiste à passer de l'état final à l'état initial en suivant une chaîne d'états intermédiaires, chaque état intermédiaire étant généré en accord avec le système de production.*

Ce type de système de production est dit à chaînage arrière.

Bien entendu, un système de production adapté à un raisonnement inductif est généralement différent d'un système de production adapté à un raisonnement déductif.

Il est souvent possible de construire, pour un problème donné, un système de production déductif et un système de production inductif. Comment choisir ? Il faut considérer le facteur de branchement de chacun des systèmes. Ainsi, pour la démonstration de théorème, un système inductif a dans beaucoup de cas un facteur de branchement inférieur à un système déductif, dont la combinatoire est explosive.

On peut également, dans certains cas, mélanger les deux stratégies, et réaliser une partie de la recherche par chaînage avant et une autre partie par chaînage arrière. C'est le

3 Il existe souvent plusieurs états finaux.

cas en démonstration de théorème, où l'on peut par exemple commencer par « décomposer » en sous-buts le théorème à démontrer (chaînage arrière), puis tenter de démontrer chacun des sous-buts à partir des axiomes (chaînage avant). Ce type de mécanisme est appelé *chaînage mixte.*

13.3.3 Monotonie et commutativité

Les systèmes de production peuvent être classés en quatre catégories à l'aide des deux critères suivants :

Définition 13.7 – Système monotonique – *Un système de production est dit monotonique*[4] *si l'application d'une règle à l'instant t laisse toutes les autres règles qui étaient applicables à t applicables à* $t' > t$.

Un exemple de système monotonique est la démonstration de théorème en raisonnement déductif. En effet, l'application d'une règle se contente de rajouter une formule dans la base de théorèmes et laisse toutes les règles qui étaient applicables encore applicables.

Définition 13.8 – Système partiellement commutatif – *Un système de production est dit partiellement commutatif s'il vérifie la condition suivante : soit un état* X *quelconque et une séquence de règles de production* $(s_1, s_2, \ldots, s_n)$ *telle que*

$$X \overset{s_1}{\rightsquigarrow} X_1 \overset{s_2}{\rightsquigarrow} X_2 \ldots \overset{s_{n-1}}{\rightsquigarrow} X_{n-1} \overset{s_n}{\rightsquigarrow} Y$$

Alors pour toute permutation σ *la séquence* $(s_{\sigma(1)}, s_{\sigma(2)}, \ldots, s_{\sigma(n)})$ *doit vérifier :*

$$X \overset{s_{\sigma(1)}}{\rightsquigarrow} X'_1 \overset{s_{\sigma(2)}}{\rightsquigarrow} X'_2 \ldots \overset{s_{\sigma(n-1)}}{\rightsquigarrow} X'_{n-1} \overset{s_{\sigma(n)}}{\rightsquigarrow} Y$$

à condition que les conditions initiales d'application des règles $s_{\sigma(i)}$ *soient satisfaites à chaque étape.*

Il faut remarquer que l'on n'impose pas que pour toute permutation la séquence $(s_{\sigma(1)}, s_{\sigma(2)}, \ldots, s_{\sigma(n)})$ soit valide. Nous verrons cela sur le problème des missionnaires et des cannibales, qui est partiellement commutatif.

Définition 13.9 – Système commutatif – *Un système de production est dit commutatif s'il est monotonique et partiellement commutatif.*

Nous avons dit qu'il était possible de définir plusieurs systèmes de production pour un même système. Il est en fait possible de construire pour tout problème un système de production commutatif qui le résolve. Mais ce système peut être totalement inefficace en terme d'espace et de temps de calcul. Nous verrons dans un prochain chapitre comment choisir un système de production adapté, en fonction des caractéristiques du problème. Examinons auparavant quelques exemples.

---

4 « monotonique » est une traduction de l'anglais adoptée par nombre d'auteurs.

13.3.4 Exemples

Nous allons reprendre les exemples du paragraphe précédent et définir pour chacun d'entre eux un système de production adapté.

Les missionnaires et les cannibales : On peut ici décrire le système de production par les règles suivantes (rappelons que X représente le nombre de missionnaires sur la rive de gauche, Y le nombre de cannibales sur la même rive et P la position de la barque) :

$X \geq 2$	$X-2 \geq Y$	:	(X,Y,G)	$\rightsquigarrow$	$(X-2,Y,D)$
		:	$(2,Y,G)$	$\rightsquigarrow$	$(0,Y,D)$
$Y \geq 2$	$3-X \geq (3-Y)+2$	:	(X,Y,G)	$\rightsquigarrow$	$(X,Y-2,D)$
$Y \geq 2$		:	$(3,Y,G)$	$\rightsquigarrow$	$(3,Y-2,D)$
$X \geq 1$	$X-1 \geq Y$	:	(X,Y,G)	$\rightsquigarrow$	$(X-1,Y,D)$
		:	$(1,Y,G)$	$\rightsquigarrow$	$(0,Y,D)$
$Y \geq 1$	$3-X \geq (3-Y)+1$	:	(X,Y,G)	$\rightsquigarrow$	$(X,Y-1,D)$
$Y \geq 1$		:	$(3,Y,G)$	$\rightsquigarrow$	$(3,Y-1,D)$
$X \geq 1$	$Y \geq 1$	:	(X,Y,G)	$\rightsquigarrow$	$(X-1,Y-1,D)$
$3-X \geq 2$	$(3-X)-2 \geq 3-Y$	:	(X,Y,D)	$\rightsquigarrow$	$(X+2,Y,G)$
		:	$(1,Y,D)$	$\rightsquigarrow$	$(3,Y,G)$
$3-Y \geq 2$	$X \geq Y+2$	:	(X,Y,D)	$\rightsquigarrow$	$(X,Y+2,G)$
$3-Y \geq 2$		:	$(0,Y,D)$	$\rightsquigarrow$	$(0,Y+2,G)$
$3-X \geq 1$	$(3-X)-1 \geq 3-Y$	:	(X,Y,D)	$\rightsquigarrow$	$(X+1,Y,G)$
		:	$(2,Y,D)$	$\rightsquigarrow$	$(3,Y,D)$
$3-Y \geq 1$	$X \geq Y+1$	:	(X,Y,D)	$\rightsquigarrow$	$(X,Y+1,G)$
$3-Y \geq 1$		:	$(0,Y,D)$	$\rightsquigarrow$	$(0,Y+1,G)$
$3-X > 1$	$3-Y > 1$	:	(X,Y,D)	$\rightsquigarrow$	$(X+1,Y+1,G)$

Muni de ces règles, il est clair que partant d'un état valide, je peux générer tous les états valides qui en découlent, et seulement les états valides. Ainsi partant de l'état initial, et appliquant à chaque étape l'ensemble des règles de production, je dois générer l'ensemble des états possibles et en particulier l'état final s'il appartient à l'ensemble des états possibles[5]. Le système de production que nous avons défini a la particularité de pouvoir être utilisé aussi bien pour un raisonnement déductif que pour un raisonnement inductif. On remarque également que ce système est non monotonique et partiellement commutatif.

La démonstration de théorèmes : Les règles d'inférence de l'arithmétique formelle constituent un système de production pour la démonstration de théorèmes. Si nous considérions un système de Post de l'arithmétique formelle, les règles de production (au sens de la théorie des langages) seraient exactement les règles de production de notre problème. Ce système de production est de type déductif, monotonique, et partiellement commutatif donc commutatif. En revanche, les mécanismes de résolution que nous avons introduits au chapitre 6 constituent un système de production inductif.

---

5 Nous ne considérons pas ici la technique utilisée pour générer ces états (c'est-à-dire pour se déplacer dans l'espace d'états), ni la méthode utilisée pour représenter le monde d'états ainsi généré. Nous aborderons plus loin ces techniques.

	Monotonique	Non-monotonique
Partiellement commutatif	Démonstration de théorème	Missionnaires et cannibales
Non partiellement commutatif	Voyageur de commerce	Poker

Table 13.1 – Caractéristiques de problèmes

Le jeu d'échecs : L'ensemble des règles de déplacement des pièces constitue un système de production du jeu d'échecs. Remarquons que ce n'est pas le seul. En fait, on peut appeler ce système de production, par analogie avec la théorie des ensembles, un système défini en compréhension. On pourrait aussi définir un système en extension, qui consisterait à stocker l'ensemble des états possibles et l'ensemble des liaisons possibles entre chaque état. Cette méthode est bien entendu prohibitive aux échecs, vu le nombre de configurations possibles, mais possible en théorie car le nombre d'états est fini. Un système de production inductif présenterait peu d'intérêt pour le jeu d'échecs, sauf dans un cas : si l'on souhaite rechercher s'il existe un moyen d'arriver à une position donnée (problème de rétro-analyse) ; ce type de problème n'a pas grand-chose à voir avec le jeu d'échecs à proprement parler.

Le problème du voyageur de commerce : Soit un état $(x_1, x_2, \ldots, x_k)$, on construit les états $(x_1, x_2, \ldots, x_k, x_{k+1})$ où x_{k+1} décrit l'ensemble des villes différentes de $x_1, x_2, \ldots, x_k$. On génère ainsi l'ensemble des états possibles, qui contient forcément la solution. Il est impossible de construire un mécanisme inductif puisqu'on ne connaît pas l'état final de façon exacte. Ce système de production est monotonique puisque le choix d'une ville a à un instant t n'empêche pas de choisir une ville b à t' si b était aussi disponible à t. En revanche, il n'est pas commutatif, car l'ordre du choix des villes influe sur la solution.

Le jeu de Poker : Le système de production est également simple : il s'agit des règles régissant les enchères et la valeur des mains. Ce système n'est ni partiellement commutatif, ni monotonique : toute enchère est irréversible et entraîne des modifications sur la suite du déroulement de la partie. Remarquons cependant que nous ne pouvons pas prévoir, à un instant donné, le chemin suivi vers la solution. Nous allons y revenir. La notion de système de production inductif est ici sans intérêt.

Dans le tableau 13.1, sont résumées les caractéristiques de nos problèmes relativement à la monotonie et à la commutativité.

Il nous reste maintenant à trouver une représentation formelle adaptée à nos espaces d'états et à dégager des méthodes de résolution aussi efficaces que possibles.

13.4 Représentation formelle

Nous allons nous intéresser maintenant à la représentation formelle de l'espace d'états construit par un système de production.

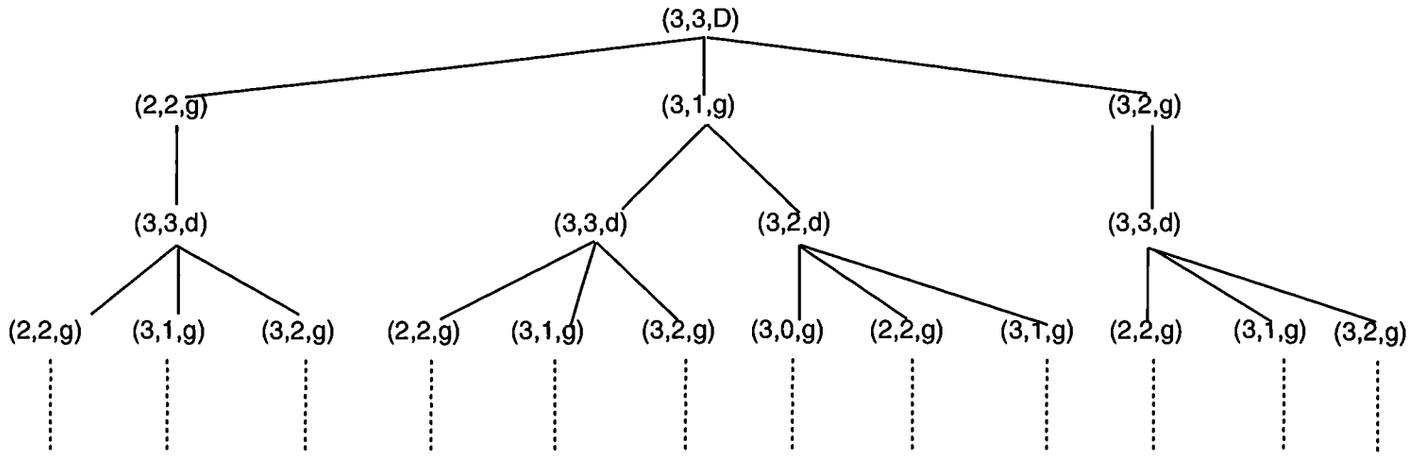


Figure 13.1 – Recherche en utilisant un arbre

13.4.1 Arbres

La méthode de représentation par arbre consiste, à chaque fois que l'on génère un état, à le relier simplement à son prédécesseur sans faire aucun test d'occurrence : on ne vérifie pas si l'état a déjà été généré.

L'avantage principal de la méthode est sa rapidité. Un test d'occurrence est toujours chose coûteuse en temps. La méthode de représentation par arbre est très utilisée en théorie des jeux, où le risque de voir une position se dupliquer est faible[6].

Ses inconvénients sont :
- la duplication d'état, qui peut entraîner un encombrement mémoire excessif ;
- le risque de ralentissement global dans la recherche de la solution : en ne s'apercevant pas que l'on génère plusieurs fois les mêmes états, on peut décrire plusieurs fois sans s'en apercevoir le même parcours. À l'extrême, on peut arriver à des cas de bouclages empêchant le système de parvenir à la solution.

Un exemple d'arbre appliqué au problème des missionnaires et des cannibales est représenté dans la figure 13.1.

13.4.2 Graphes

La méthode des graphes est similaire à celle des arbres. Chaque nouvel état généré est stocké, mais on effectue avant le stockage un test d'occurrence : si l'état a déjà été généré, on se contente de construire un arc indiquant le rebouclage.

Les avantages et les inconvénients de la méthode des graphes sont la duplication en creux des avantages et des inconvénients de la méthode des arbres. Un exemple de recherche en utilisant un graphe pour résoudre le problème des missionnaires et des cannibales est représenté figure 13.2.

---

6 Pas dans tous les cas cependant. Ainsi, les meilleurs programmes d'échecs traitent différemment les finales, voir chapitre 15.

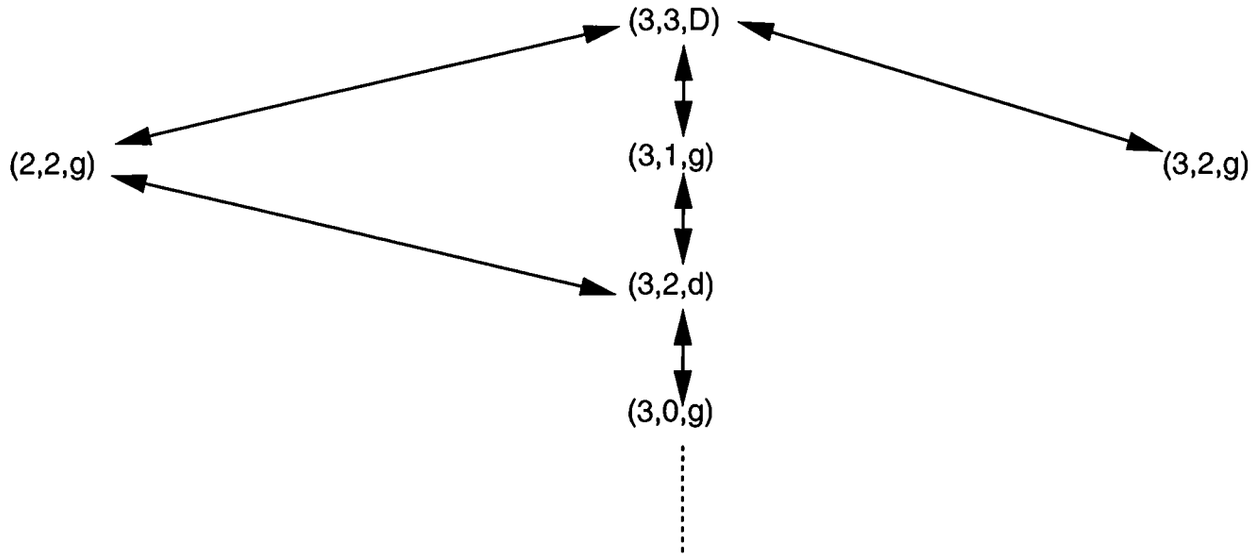


Figure 13.2 – Recherche en utilisant un graphe

13.4.3 Graphes ET-OU

Dans les graphes tels que nous les avons présentés tout nœud suivi d'un certain nombre d'arcs signifie que l'on peut passer de ce nœud à n'importe lequel des nœuds qui lui sont connectés[7]. On parle alors de graphe (ou d'arbre) *OU*.

Cependant, dans certaines catégories de problèmes, il est parfois utile d'utiliser une représentation plus complexe, appelée représentation *ET-OU*. Dans ce type de représentation, il part toujours plusieurs arcs de chaque nœud, mais certains arcs sont reliés entre eux. Cela signifie que pour résoudre le nœud courant, il faut résoudre tous les sous-nœuds auxquels il est raccordé par un de ces « paquets d'arcs »[8] ou hyper-arcs.

Voici un exemple de représentation d'arbre de résolution *ET-OU* en démonstration automatique de théorème appliquée à la formule $p \wedge ((q \vee (r \wedge s))$ en utilisant un système de production inductif (figure 13.3).

13.5 Stratégies de résolution

Il s'agit ici d'examiner les différentes stratégies permettant de parcourir un arbre de la façon la plus efficace possible.

13.5.1 Algorithme du British Museum

À chaque étape, on génère un seul nœud en choisissant au hasard une règle de production. À partir de ce nouveau nœud on itère le processus jusqu'à l'obtention d'une solution. Si, à un instant donné, on aboutit à une impasse (plus de règles de production sélectionnables), on recommence l'opération depuis la racine de l'arbre.

---

7 Dit autrement : on peut considérer qu'un nœud mène à une solution si le premier fils *ou* le deuxième fils *ou* ... le n-ième fils mène à la solution.

8 Ou encore : un nœud ne mène à la solution que si l'ensemble de ses fils mène à une solution.

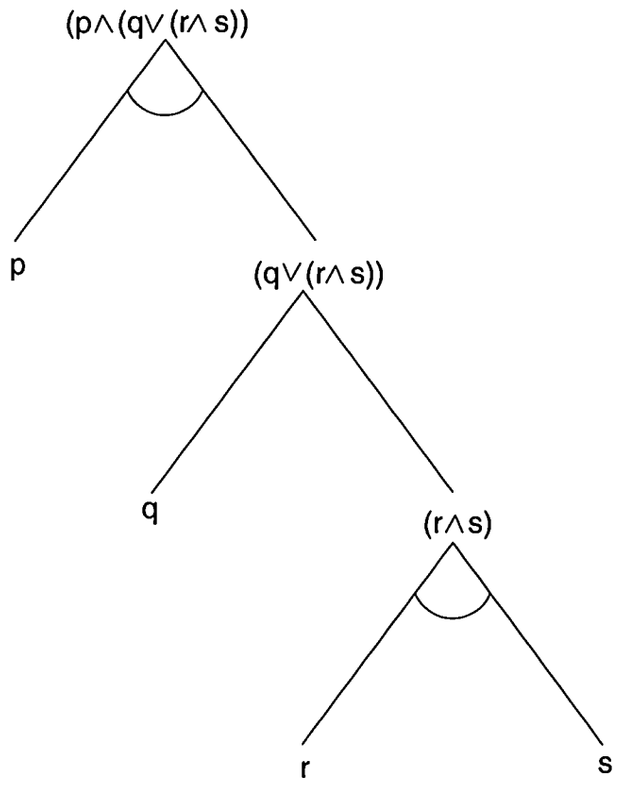


Figure 13.3 – $p \wedge ((q \vee (r \wedge s))$ représenté par un arbre *ET-OU*

Cet algorithme[9] est extrêmement inefficace. Il ne garantit même pas qu'une solution sera trouvée.

13.5.2 Recherche en profondeur et retour-arrière

Il s'agit d'une implantation plus intelligente de l'algorithme précédent[10]. À chaque échec, on réalise un retour au nœud précédant le nœud où l'on a constaté l'impasse et on choisit une autre règle de production parmi celles qui n'ont pas été encore utilisées. Celle-ci générera ainsi un autre nœud. Nous voyons sur la figure 13.4 une recherche dans un arbre en utilisant cet algorithme. Le but de la recherche est de trouver une position dont la valuation est supérieure à 50. Les chiffres en italique représentent la valuation des nœuds et les chiffres gras le numéro de l'étape à laquelle l'algorithme génère le nœud.

Quelques remarques s'imposent. Tout d'abord, cet algorithme peut, dans un cas défavorable, nous obliger à parcourir l'intégralité de l'espace d'état avant de trouver la solution. Il est incapable de profiter de la structure du problème pour éviter certains chemins absurdes. Enfin, il ne garantit d'aboutir à la solution que si l'on effectue des tests d'occurrence pour éviter d'éventuels bouclages. Sinon, il peut rentrer dans un cycle et ne plus en sortir, ce qui entraînera le développement d'une branche infinie dans l'arbre de recherche.

On peut appliquer ce type de mécanisme sur des problèmes comprenant un espace d'états réduit, comme par exemple le problème des missionnaires et des cannibales.

---

9 Le nom de l'algorithme vient du paradoxe suivant : si on laissait des singes devant des machines à écrire, ils réaliseraient une implantation de cet algorithme, et produiraient, si on leur laissait assez de temps, tous les livres de la bibliothèque du musée (preuve statistique).

10 En anglais : *depth-first.*

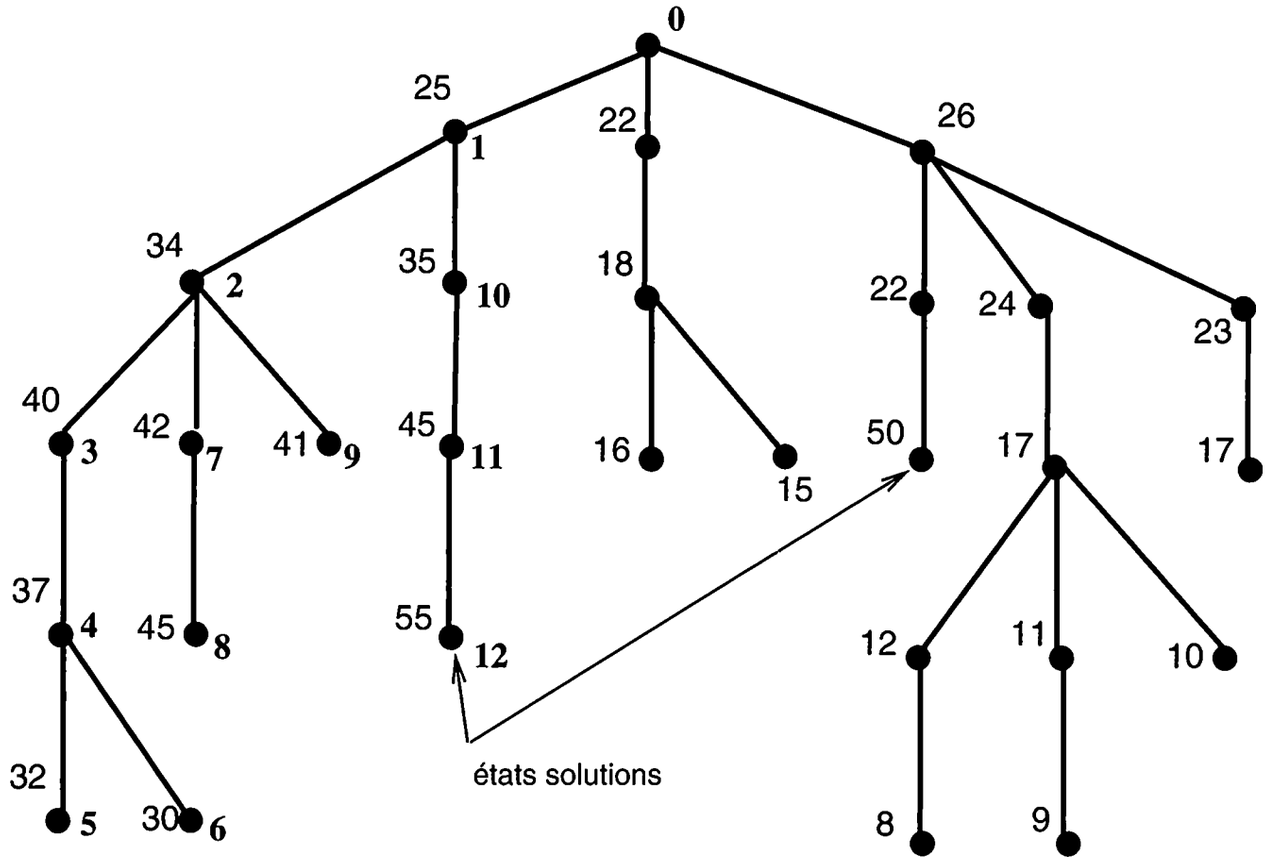


Figure 13.4 – Recherche en profondeur et retour-arrière

Il peut aussi être appliqué pour prouver l'inconsistance d'une formule sous forme clausale de Horn en calcul propositionnel ou en logique du premier ordre. C'est d'ailleurs la méthode utilisée par la plupart des langages PROLOG[11].

13.5.3 Recherche en largeur

La recherche en largeur consiste à générer l'ensemble des nœuds niveau par niveau. L'avantage de la recherche en largeur est qu'elle garantit de trouver une solution, même s'il existe un cycle. D'autre part, cette solution sera également la moins profonde. Son inconvénient majeur est sa grande consommation de mémoire[12]. Comme la recherche en profondeur, la recherche en largeur peut, dans des cas défavorables, être amenée à considérer l'intégralité de l'arbre et ne sait pas tenir compte de la structure du problème pour améliorer la recherche (figure 13.5).

13.5.4 La notion d'heuristique

Une heuristique[13] a pour but de diriger la recherche dans l'arbre, de façon à réduire le temps de résolution du problème ; elle introduit des mécanismes qui dérivent de connaissances spécifiques au problème.

---

11 Malheureusement, cette technique rend la résolution PROLOG incomplète (voir chapitre 18).

12 Un moyen terme intéressant est la recherche par « approfondissement itératif » : on effectue une recherche en profondeur d'abord limitée à la profondeur 1, puis 2... jusqu'à résolution. Le coût mémoire est fortement réduit alors que le temps machine est asymptotiquement du même ordre (Cf. (Korf 1985)).

13 *heuristique* : emprunté au grec *euristikê, (tekhnê)*, [art] de découvrir. *Dictionnaire étymologique Larousse.*

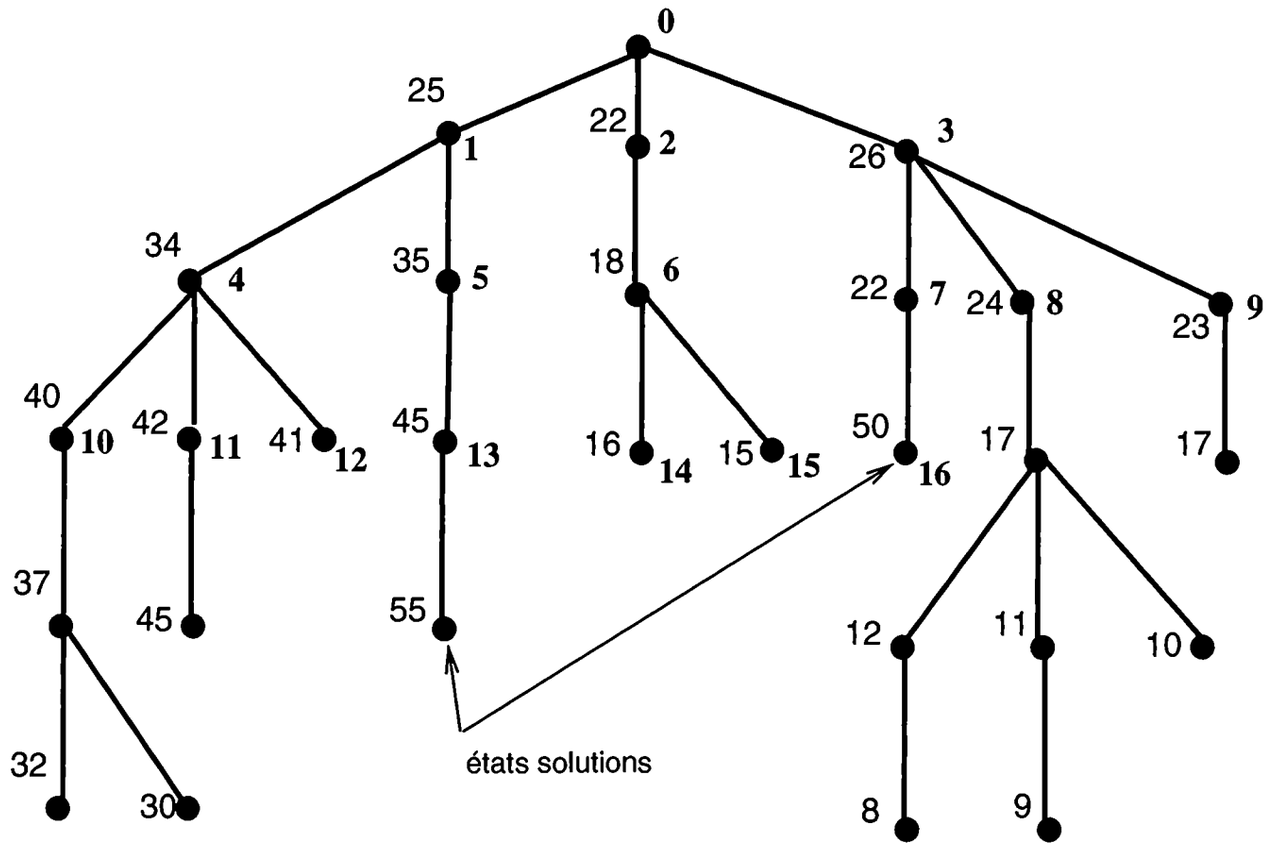


Figure 13.5 – Recherche en largeur

Il est commun de combiner l'utilisation d'une heuristique avec des méthodes de recherche ne garantissant plus la complétude de la résolution, afin d'améliorer encore l'efficacité. Ce type de technique est justifiable, car, bien souvent, on ne cherche pas la meilleure solution à un problème mais simplement une solution satisfaisante : une personne cherchant à établir un emploi du temps ne cherche pas le meilleur emploi du temps possible , mais un bon compromis entre qualité et temps de construction. Ce type d'exemple est très général : il va de même lorsqu'on cherche une démonstration de théorème, un circuit passant par plusieurs villes ou une place de parking.

Malheureusement, introduire une heuristique ne suffit pas toujours à résoudre un problème, même de façon à peu près satisfaisante. Tout dépend de la complexité du problème et de la qualité de l'heuristique. Nous examinerons de façon plus formelle différents types d'heuristiques en présentant les algorithmes de type A.

La notion de *recherche heuristique* est une des notions de base de l'intelligence artificielle.

13.5.5 L'escalade

L'escalade[14] est une méthode de recherche en profondeur dans laquelle on introduit une fonction heuristique pour choisir à chaque étape le nœud à générer, au lieu de générer un nœud en prenant une règle de production au hasard.

---

14 *hill-climbing* en anglais.

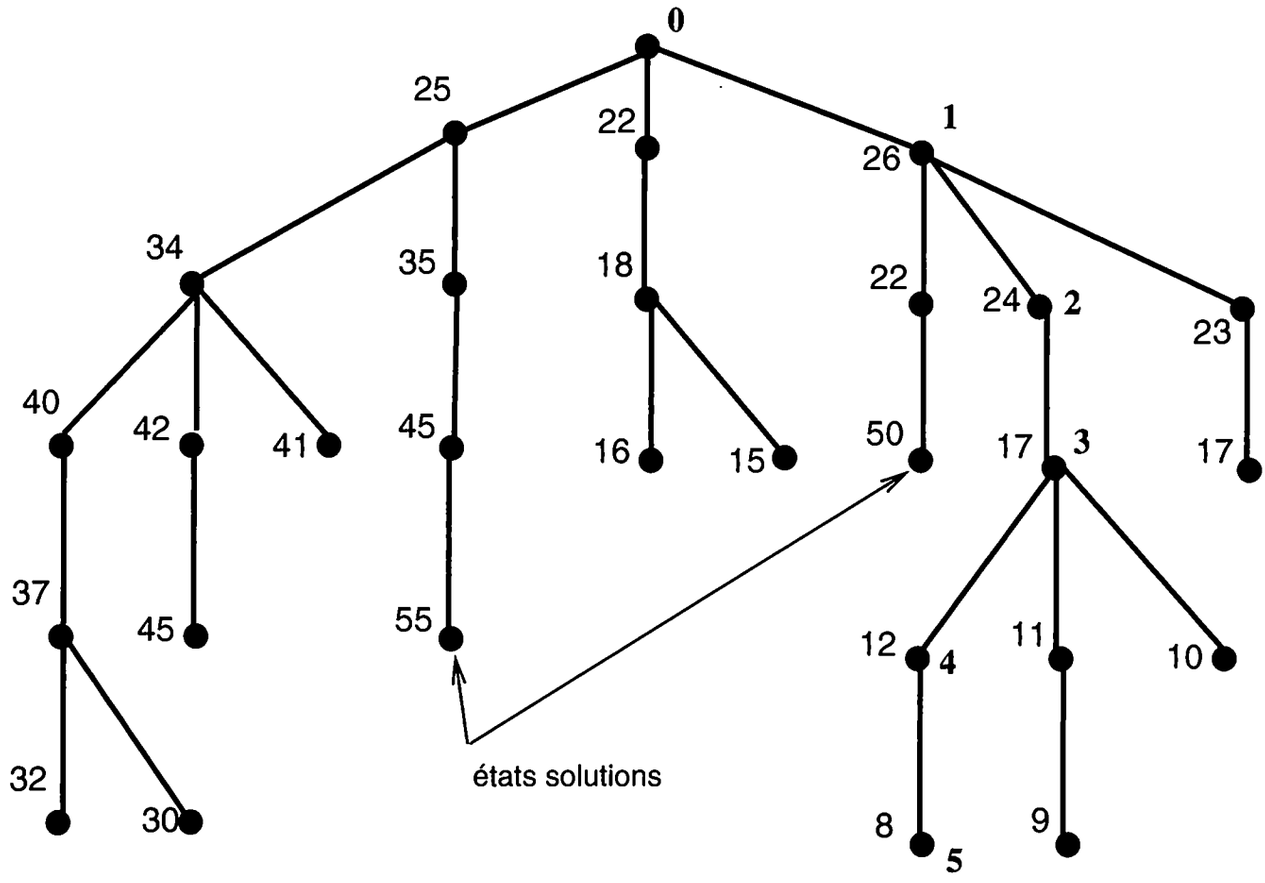


Figure 13.6 – Escalade et recherche en profondeur

Ainsi, considérons le problème du voyageur de commerce. Si nous générons les nœuds en utilisant l'algorithme de recherche en profondeur avec retour-arrière, il se peut que nous générions $n!$ nœuds.

Si nous choisissons comme heuristique de générer à chaque fois le nœud qui ajoute à la liste des villes déjà visitées la ville la plus proche de la dernière traversée, notre algorithme a une complexité en : $n + (n-1) + \cdots + 2 + 1 = \frac{n(n+1)}{2}$ donc une complexité polynomiale. Certes, la solution trouvée n'est pas forcément la meilleure[15], et il nous faudrait poursuivre la recherche et générer autant de nœuds que précédemment pour trouver la meilleure solution. Mais la solution approchée apparaît comme admissible à la plupart des gens et peut donc, bien souvent, être retenue.

La recherche en escalade peut aussi être appliquée classiquement (figure 13.6). Dans le cas de notre problème, c'est la valuation d'un nœud qui sert de fonction heuristique. Comme le montre la figure, le résultat donné par « escalade » dans ce cas n'est même pas une solution convenable. L'escalade reste une technique intéressante dans le cas de systèmes commutatifs.

---

15 On a même pu voir, au chapitre 11, que pour ce problème, il est impossible de borner la perte de qualité qui résulte de l'emploi d'un algorithme polynomial, *qu'il soit d'origine IA ou non.*

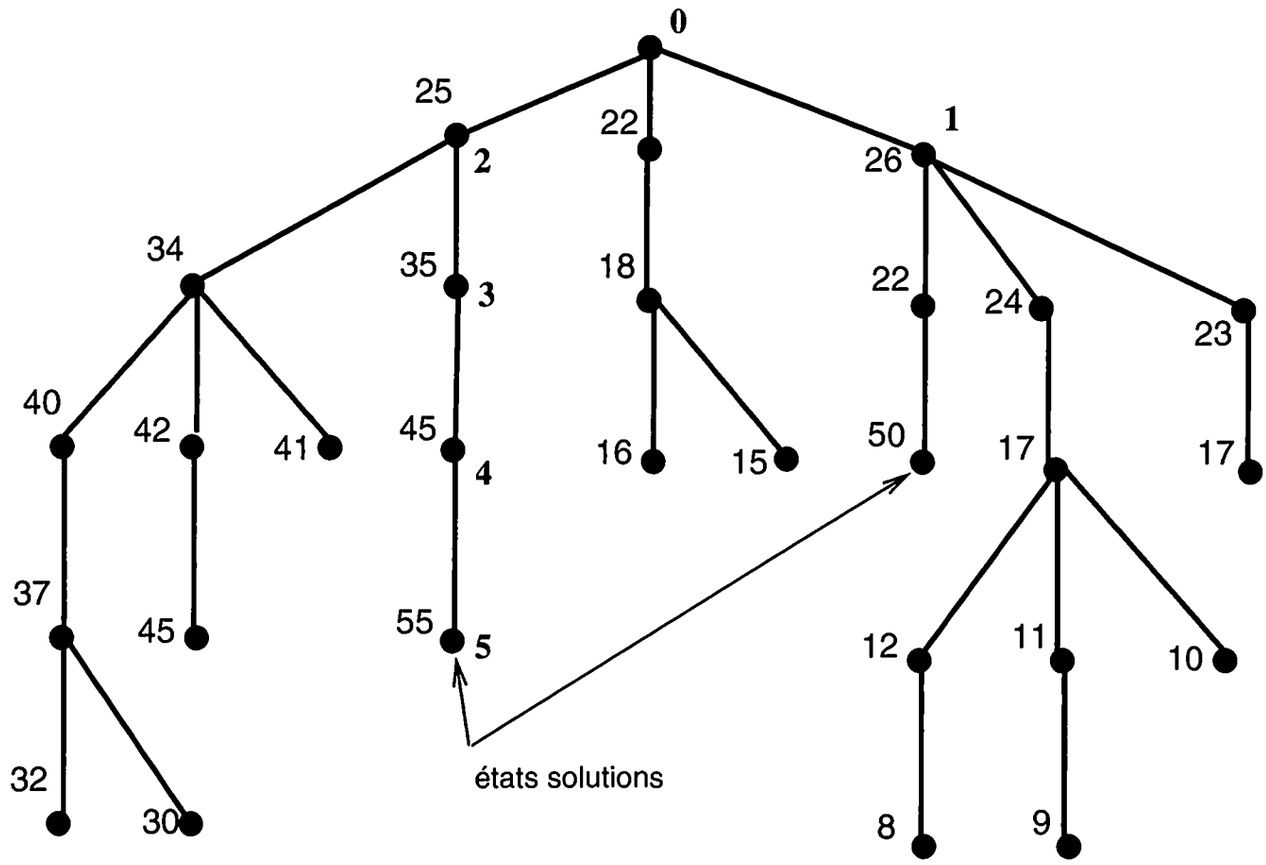


Figure 13.7 – Recherche meilleur en premier

13.5.6 Recherche *meilleur en premier* : graphe OU

La recherche *meilleur en premier*[16] est un compromis entre une recherche en largeur et une recherche en profondeur, en utilisant une fonction heuristique pour guider la génération des nœuds.

Le mécanisme est le suivant : on développe à chaque étape le meilleur nœud encore non développé. On inclut les nœuds ainsi générés dans l'ensemble des nœuds connus et on recommence le processus (figure 13.7). Nous allons présenter un raffinement de cette méthode de façon plus formelle dans le cadre des graphes OU et des problèmes de minimisation de coût additif : l'algorithme[17] A*, chargé de calculer le plus court chemin menant d'un état initial à un état final.

Les données du problème seront :

u_0 : l'état initial.

T : l'ensemble des états terminaux.

$\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$: l'ensemble des règles de production.

$h(u)$: u est un état, h est une fonction heuristique qui estime le coût de passage de u à l'état final.

$k(u, v)$: u et v sont des états, k est une fonction qui donne le coût du passage par l'arc (u, v).

Nous utiliserons comme variables annexes :

---

16 *best-first* en anglais.

17 Cet algorithme a été présenté pour la première fois dans (Hart *et al.* 1968).

D : une liste contenant l'ensemble des états déjà développés (un état est développé si on a déjà généré l'ensemble de ses successeurs).

G : une liste contenant l'ensemble des états générés non encore développés.

$g(v)$: un tableau indexé par les états qui contient le coût du trajet pour atteindre l'état v.

$f(v)$: un tableau indexé par les états qui contiendra pour chaque état le coût total estimé.

père(v) : un tableau indexé par les états qui donne, pour chaque état, l'état qui l'a généré.

Nous utiliserons également deux fonctions annexes, *premier*(G) qui renverra le premier élément de G, et *insérer-selon-f*(G, v) qui insérera v dans G dans l'ordre f croissant puis g décroissant.

Il ne nous reste plus qu'à présenter l'algorithme :

1. $G \leftarrow (u_0)$; $D \leftarrow \varnothing$; $g(u_0) \leftarrow 0$; $f(u_0) \leftarrow 0$
2. **Tant que** $G \neq \varnothing$ **faire**
   (a) $u \leftarrow$ *premier*(G) ; $G \leftarrow G \setminus (u)$; $D \leftarrow D \cup (u)$
   (b) **Si** $u \in T$ **alors** imprimer_solution ; **fin_programme** ; **fin_si**
   (c) **Pour** i **allant de** 1 **jusqu'à** n **faire**
      i. $v \overset{p_i}{\leftarrow} u$
      ii. **Si** $[v \notin D \cup G]$ **ou** $[g(v) > g(u) + k(u, v)]$ **alors**
      iii. $g(v) \leftarrow g(u) + k(u, v)$
      iv. $f(v) \leftarrow g(v) + h(v)$
      v. *père*$(v) \leftarrow u$
      vi. *insérer-selon-f*(G, v)[18]
      vii. **fin_si**
   (d) **fin_faire**
3. **fin_faire**

Quelques remarques s'imposent.

- la fonction $k(u, v)$ peut être prise identiquement nulle si on ne se préoccupe pas de la longueur du chemin menant à la solution. Si on cherche le chemin comprenant le moins d'arcs possibles, on prendra $k(u, v) = 1$ pour tout arc (u, v).
- la fonction $h(u)$ est une heuristique qui estime la distance entre l'état courant et l'état final. De façon formelle, $h(u)$ tente d'estimer le minimum sur tous les chemins $(u, v_1, \ldots, v_n, u_t)$, où u_t est un état terminal, de la somme :

$$k(u, u_1) + k(u_1, u_2) + \cdots + k(u_n, u_t)$$

La valeur réelle de ce minimum est notée $h^*(u)$. On dégage plusieurs catégories de fonction heuristique h.

---

18 Nous rappelons au lecteur que la fonction *insérer-selon-f* utilise la fonction (ou tableau) f pour insérer v à la bonne position dans la liste G.

Définition 13.10 – Heuristique parfaite – *Une heuristique h est dite parfaite si et seulement si :*

$$\forall u, v, h(u) = h(v) \Leftrightarrow h^*(u) = h^*(v)$$

Définition 13.11 – Heuristique presque parfaite – *Une heuristique h est dite presque parfaite si et seulement si :*

$$\forall u, v, h(u) < h(v) \Rightarrow h^*(u) < h^*(v)$$

Définition 13.12 – Heuristique monotone – *Une heuristique h est dite monotone (ou consistante) si et seulement si :*

$$\forall u, \forall v \textit{ descendant de } u, h(u) - h(v) \leq k(u, v)$$

Définition 13.13 – Heuristique minorante – *Une heuristique h est dite minorante (ou admissible) si et seulement si :*

$$\forall u, h(u) \leq h^*(u)$$

- si l'heuristique h est parfaite, alors l'algorithme convergera immédiatement vers le but ;
- si h est une heuristique minorante alors l'algorithme A^* trouvera toujours le chemin optimal ;
- une heuristique monotone est non seulement admissible, mais garantit, dans le cadre de l'algorithme A^*, que pour tout nœud u développé ($u \in D$), le chemin calculé est optimal ($g(u)$ est effectivement égal au coût du plus court chemin menant à u).

On s'est également intéressé à la complexité de A^*.

- dans le pire des cas, la complexité est en 2^N, où N est le nombre de nœuds du graphe, puisque chaque état u sera développé autant de fois qu'il y a de chemins de u_0 à u ;
- si h est une heuristique minorante, on peut, en modifiant légèrement l'algorithme, avoir une complexité en N^2 ;
- Si h est monotone, la complexité est linéaire.

Cependant, il faut remarquer que même une complexité linéaire sur le nombre d'états est clairement inutilisable dans la plupart des cas pratiques. Ainsi, le problème du voyageur de commerce a un nombre d'états en $N = n!$ où n est le nombre de villes.

On peut s'intéresser à un autre type de complexité, celle du nombre de nœuds que l'on considérera en fonction du nombre d'arcs entre l'état initial et un état final, que nous noterons M. Quelques résultats connus sont :

- si h est presque parfaite, alors la complexité est linéaire en M ;
- si h est monotone, en considérant un arbre dont le facteur de branchement est K, la complexité est exponentielle en K^M ;

– Pour une heuristique minorante, toujours pour un arbre K-aire, on peut déterminer la complexité de l'algorithme en fonction de l'écart entre h et h^* :

Erreur	Complexité
$(1-r)h^* \leq h$	$K^{\alpha M}$
$h^* - \sqrt{h^*} \leq h$	$\sqrt{M}K^{\sqrt{M}}$
$h^* - \log h^* \leq h$	$M^{\log K}$
$h^* - r \leq h$	MK^r

Enfin, notons qu'un certain nombre de résultats à caractère général montre que si on ne dispose pas d'une « très bonne » heuristique (cas le plus fréquent), l'algorithme A^* développera en moyenne un nombre exponentiel d'états en fonction de la longueur optimum du chemin. Pour un énoncé plus détaillé de toutes ces propriétés on peut se reporter à (Farreny and Ghallab 1987) ou à (Pearl 1990). Les habitués des algorithmes sur les graphes reconnaîtrons dans l'algorithme A* une variante de l'algorithme de Dijkstra (ce dernier n'utilisant pas d'heuristique).

L'algorithme A^* est extrêmement populaire malgré les résultats que nous venons d'énoncer. En effet, il est général, simple à implanter et amplement suffisant pour nombre de problèmes de taille réduite. D'autre part, on sait qu'il est « optimal », c'est-à-dire qu'il fait « au mieux » avec les informations dont il dispose. Pour résoudre les problèmes de consommation mémoire que l'A^* fait apparaître avec facilité, des algorithmes de recherche en approfondissement itératif (*iterative deepening*, assez semblables à ceux construits au-dessus de l'algorithme α-β) ont été développés (Korf 1985; Sarkar *et al.* 1991).

Enfin, signalons le développement d'algorithme à seuil, ou ε-admissible. Ces algorithmes ne cherchent pas forcément la meilleure solution, mais une solution dont l'écart relatif par rapport à l'optimum est inférieur à ε. Il existe ainsi un algorithme A^*_ε qui applique ce principe à l'algorithme A^* (cf. (Farreny and Ghallab 1987; Pearl 1990)).

13.5.7 Recherche *meilleur en premier* : graphe ET-OU

L'algorithme A^*, que nous venons de présenter, ne peut s'appliquer dans des graphes ET-OU. En effet, nous ne devons plus considérer des chemins de moindre coût, mais des groupes de chemins, puisque pour résoudre un nœud, il faut résoudre plusieurs sous-nœuds.

Une extension de l'algorithme A^* a été définie dans ce but. Il s'agit de l'algorithme AO^*[19]que nous ne présenterons pas (voir (Farreny and Ghallab 1987) ou (Pearl 1990)).

13.6 Analyse d'un problème

Nous allons tenter dans ce paragraphe de dégager les éléments majeurs qui caractérisent un problème[20]. L'analyse des caractéristiques d'un problème permet de choisir de

19 Cet algorithme a été présenté pour la première fois dans (Martelli and Montanari 1978).

20 Nous reprenons pour partie la classification de (Rich 1987), simplement nous ne nous intéressons pas aux systèmes utilisant des bases de connaissances importantes ou contradictoires, car nous avons estimé que ce type de problème

façon adéquate le système de production ainsi que le type de représentation formelle et la stratégie de résolution.

13.6.1 Calculabilité et complexité du problème

Nous avons consacré un chapitre entier à la complexité et à la calculabilité, nous supposons acquises les principales notions.

Un problème non-calculable ne peut pas être résolu *dans le cas général*, que ce soit par des méthodes heuristiques ou par d'autres méthodes. Un problème polynomial peut (presque) toujours être résolu par des stratégies de parcours classiques (largeur ou profondeur). Pour un problème NP-complet, il est bon d'examiner avec attention dans quel cadre nous souhaitons utiliser notre programme. De façon générale, les méthodes heuristiques peuvent apporter un gain important en terme de temps de résolution (par rapport à la force brute non guidée), mais seul le renoncement à la résolution d'un problème NP-complet (recherche d'une solution seulement « proche » de l'optimum, satisfiabilité partielle) peut permettre d'aboutir à une complexité raisonnable dans le cas général et sur des tailles importantes (cf. chapitre 11).

Examinons chacun de nos cinq problèmes :

Les missionnaires et les cannibales : Le nombre d'états de ce problème est faible. Il est possible de le résoudre par une stratégie de recherche en largeur dans un graphe.

La démonstration de théorème : En stratégie déductive, des méthodes heuristiques doivent être employées pour tenter de limiter l'explosion combinatoire. En stratégie inductive sur des clauses de Horn, il est possible d'utiliser une stratégie de recherche en profondeur.

Le jeu d'échecs : Il est évident que la complexité du problème (cf. chapitre 11) est telle que des méthodes particulières devront être utilisées.

Le voyageur de commerce Le problème est connu pour être NP-complet et n'avoir aucune ε-approximation. Il est donc évident que pour un nombre important de villes une méthode approchée devra être employée.

Le jeu de poker : Comme pour le jeu d'échecs, des stratégies adaptées doivent être utilisées en raison de l'importante complexité du problème.

13.6.2 Problèmes décomposables

Un problème est décomposable si on peut le réduire en un ensemble de problèmes plus petits que l'on essaiera alors de résoudre *séparément*. Pouvoir décomposer un problème est un grand avantage, car cela réduit considérablement la complexité de la résolution.

Nous allons examiner chacun de nos cinq problèmes sous cet aspect :

Les missionnaires et les cannibales : Le problème n'est clairement pas décomposable. Résoudre le problème avec deux missionnaires ou deux cannibales au lieu de trois ne permet pas de résoudre le problème avec trois individus.

---

relevait plus des systèmes experts que des techniques de résolution que nous présentons dans ce chapitre. De même, nous ne nous sommes guère intéressés aux problèmes d'interaction avec l'opérateur que nous développerons plus en détail lorsque nous parlerons des systèmes experts.

La démonstration de théorèmes : C'est un excellent exemple de problèmes décomposables. Supposons en effet que nous ayons à démontrer $p \wedge (q \vee (r \wedge s))$. Nous pouvons réaliser une première décomposition en deux sous-problèmes : la démonstration de p et de $q \vee (r \wedge s)$. Mais nous pouvons alors développer la seconde expression en $(q \vee r) \wedge (q \vee s)$. Ce sous-problème se décompose à son tour en deux autres sous-problèmes, $q \vee r$ et $q \vee s$. De façon générale tout $\wedge$ permet de décomposer un problème en sous-problèmes.

Le jeu d'échecs : Le jeu d'échecs n'est pas décomposable. Cependant, nous verrons que certaines techniques de résolution réalisent des décompositions d'un problème général en sous-problèmes (composition de plans stratégiques de résolution). Cet exemple montre que la barrière n'est pas toujours aussi nette qu'on peut le penser.

Le problème du voyageur de commerce : Ce problème n'est pas décomposable. Le résoudre pour quatre villes données ne permet pas de calculer plus simplement la solution si l'on rajoute une cinquième ville.

Le jeu de Poker : Clairement non décomposable.

Si un problème est décomposable, on aura souvent intérêt à choisir une stratégie de résolution par chaînage arrière, ou du moins à faire intervenir une telle stratégie. D'autre part une technique de représentation par arbres ou graphes ET-OU est particulièrement adaptée.

13.6.3 Problèmes prévisibles

Un problème est dit *prévisible* (ou encore « à information totale ») si l'ensemble des données relatives à la résolution est parfaitement connu. Si un problème n'est pas prévisible, les stratégies de résolution doivent inclure des méthodes probabilistes de raisonnement. Examinons nos cinq problèmes :

Les missionnaires et les cannibales : prévisible.

La démonstration de théorème : prévisible.

Le jeu d'échecs : prévisible.

Le voyageur de commerce : prévisible.

Le jeu de poker : non prévisible. En effet, on ne peut connaître, à un instant donné, le jeu de son (ou ses) adversaire(s). Ainsi, nous ne pouvons construire de façon certaine une solution. Tout au plus pouvons-nous prévoir avec une certaine probabilité quel type de séquence il nous faut suivre. De plus, il nous faudra à chaque enchère nouvelle, recalculer les probabilités et reconsidérer le plan à exécuter.

Les problèmes non prévisibles sont des problèmes extrêmement délicats à résoudre. Il ne faut jamais espérer trouver des solutions certaines et toujours prévoir des stratégies de contrôle évolutives. Les techniques de parcours de graphes telles que nous les avons présentées sont souvent inutilisables.

13.6.4 Problèmes ignorables ou récupérables

On dit qu'un problème est *ignorable* si l'application d'une règle du système de production produisant un état inutile n'influencera en rien la suite du problème. Le problème est dit *totalement récupérable* si la génération d'un état inutile pourra être annulée par la

suite par une autre règle du système de production. Le système est *partiellement récupérable* si nous devons implanter dans la structure de contrôle une technique consistant à revenir avant le point incorrect. Dans tout autre cas, le problème est dit irrécupérable. Remarquons que le problème de la récupération se pose surtout pour les stratégies de contrôle en profondeur, jamais pour les stratégies en largeur.

Reprenons une fois de plus nos cinq problèmes :

Les missionnaires et les cannibales : Le problème n'est pas ignorable mais il est totalement récupérable. En effet, si j'applique une règle i déplaçant des missionnaires ou des cannibales, je peux appliquer la règle j inverse de la règle i (c'est-à-dire celle qui replace le problème dans sa forme initiale).

La démonstration formelle : Il s'agit d'un exemple de problème ignorable. En effet, si je génère, lors de ma démonstration, un résultat inutile, cela n'a strictement aucune importance.

Le jeu d'échecs : Si le but de notre programme est de trouver un état final à partir d'un état initial (problème de mat), le système est non-ignorable et partiellement récupérable. En effet, il nous faut implanter dans la structure de contrôle la notion de retour en arrière pour pouvoir éliminer l'étape incorrecte dans notre résolution.

Si notre programme d'échecs doit, non plus résoudre des mats, mais bien jouer aux échecs, alors, pour des raisons de complexité, il est clair que le but n'est plus de calculer un état final, mais bien de jouer à chaque étape un coup aussi bon que possible. Le problème est que tout coup réellement joué ne peut plus être annulé : le problème est irrécupérable.

Le voyageur de commerce : Le problème du voyageur de commerce est un problème non-ignorable et récupérable.

Le jeu de poker : C'est un exemple parfait de système irrécupérable. Toute enchère effectuée ne peut être reprise.

Nous avons vu que la notion de problème récupérable est souvent liée à la technique de résolution et à la stratégie de contrôle. De façon générale, des problèmes que nous n'espérons pas résoudre en une étape, et qui se dérouleront avec une interaction avec le milieu extérieur (échecs, poker), sont toujours irrécupérables.

Les problèmes ignorables sont toujours des problèmes que l'on résoudra par des systèmes de production commutatifs.

13.6.5 Optimisation ou satisfiabilité

Comme le fait remarquer (Simon and Kadane 1975), il faut distinguer soigneusement les problèmes d'optimisation (trouver la meilleure solution), des problèmes de satisfiabilité (trouver une solution qui satisfait à).

Ainsi dans les cinq cas que nous étudions :

Les missionnaires et les cannibales : Problème de satisfiabilité, nous nous contentons évidemment d'une solution (on préférait probablement la plus courte, mais rien ne l'indique dans l'énoncé du problème).

La démonstration formelle : Problème de satisfiabilité. Nous nous préoccupons assez peu de trouver la « meilleure » démonstration, d'autant qu'il est assez délicat de définir ce qu'est une « bonne » démonstration[21].

Le jeu d'échecs : S'il s'agit de résoudre des problèmes de mat, on peut considérer qu'il s'agit d'un problème de satisfiabilité, bien que dans certains cas, le nombre de coups pour mater soit minimisé. S'il s'agit au contraire de jouer aux échecs, le problème est un problème d'optimisation (trouver le meilleur coup possible à un instant donné)[22].

Le voyageur de commerce : Il s'agit d'un problème d'optimisation, mais on ne connaît aucune méthode efficace pour le résoudre. On le transforme donc en un problème de « semi-satisfiabilité » en relâchant les contraintes sur l'optimisation.

Le jeu de poker : Il s'agit bien entendu d'un problème d'optimisation.

13.7 Conclusion

Nous n'avons fait que rapidement présenter les principales stratégies d'exploration. Ces techniques, très générales, ont eu des emplois très variés en intelligence artificielle (preuve de théorèmes, satisfaction de formules, de contraintes, systèmes experts) et en recherche opérationnelle (*branch and bound...*).

De nombreuses sophistications, aux implications importantes, ont été apportées depuis quelques années aux algorithmes de base présentés ici. Le lecteur intéressé pourra se reporter à (Pearl 1990; Farreny and Ghallab 1987) et aux nombreuses publications sur le sujet (Korf 1985; Dechter and Pearl 1988b; Kanal and V.Kumar 1988; Korf 1988; 1990; Sarkar *et al.* 1991; Korf 1987).

---

21 On pourrait rechercher une démonstration comprenant un nombre minimal d'inférences, mais d'autres critères entrent en jeu aux yeux du mathématicien.

22 Les deux problèmes sont tellement différents que les ordinateurs d'échecs disposent de techniques différentes pour les résoudre. S'il était envisageable de le résoudre, on pourrait, dès le départ, se poser le problème en terme de satisfiabilité : « trouver un coup gagnant ». Ce problème étant trop difficile, on se limite à chercher le coup qui semble le meilleur.

CHAPITRE 14

Problèmes de satisfaction de contraintes

Thomas Schiex†

14.1 Introduction

Les problèmes de satisfaction de contraintes[1] regroupent sous leur houlette une partie des problèmes de l'intelligence artificielle et de la recherche opérationnelle. Dans le premier cas, l'instance la plus caractéristique est le problème de satisfiabilité d'un ensemble de clauses de la logique propositionnelle[2] ; dans le second, on pourra faire référence aux problèmes d'ordonnancement.

De façon très générale, une contrainte correspond à l'énoncé d'une propriété relative à différents objets. Ainsi, l'équation « $x.y + z = y$ » met en relation les variables x, y et z en limitant les valeurs que peuvent prendre simultanément ces objets. La résolution d'un problème de *satisfaction* de contraintes revient à trouver des valeurs dans les domaines des variables x, y et z qui vérifient l'équation *i.e.*, qui satisfassent la contrainte. Les contraintes ont un domaine d'utilisation très vaste : représentation de lois physiques ($U = R.I$, $F = \frac{1}{2}\rho S v^2 \ldots$), de connaissances symboliques (logique propositionnelle), de propriétés variées (géométriques, thermiques...), etc.

Il existe donc une très grande variété de types de contraintes que l'on peut tenter de classifier, entre autres, suivant :

- la nature des domaines dans lesquels les objets peuvent prendre leurs valeurs (ensemble de symboles, intervalle ou partie de $\mathbb{R}$, de $\mathbb{N}$, intervalle temporel...) ;
- le langage dans lequel elles sont exprimées (contraintes linéaires, quadratiques, exprimées en extension...) ;
- les situations qu'elles décrivent habituellement (contraintes temporelles, géométriques...).

---

† Je suis redevable, pour la rédaction de ce chapitre, à tous les membres du projet PRC-IA (Projet CSPFlex. *et al.* 1992) « Représentation et traitement de la flexibilité dans les problèmes sous contraintes ». Le lecteur retrouvera, lors de la lecture, des références aux travaux qui m'ont inspiré. Une « omission » essentielle que vient combler cette note : le « tutoriel » conçu par l'équipe du LIRMM (et présenté par M.C. Vilarem) dans le cadre du PRC-IA (Vilarem *et al.* 1991; Schiex and Vilarem 1992).

1 Référencés le plus souvent sous le nom de CSP pour *Constraint Satisfaction Problems*.

2 Ce problème (SAT) est le premier de la longue série des problèmes NP-complets recensés par exemple dans (Garey and Johnson 1979). Voir aussi le chapitre 11.

La nature d'une contrainte n'est pas directement liée à son mode d'expression. En particulier, et dans le cas de domaines finis, il est toujours possible de ramener une contrainte exprimée via un prédicat à une contrainte exprimée en extension, via l'ensemble des n-uplets de valeurs qu'elle autorise ou qu'elle interdit. La finitude est une garantie (certes, un peu forte) de décidabilité pour le problème d'appartenance.

Naturellement, la nature des contraintes affecte aussi la difficulté du problème de satisfaction. Dans le domaine de la programmation linéaire, il existe des algorithmes de complexité polynomiale[3] pour résoudre des problèmes de satisfaction ou même d'optimisation d'une fonction linéaire sous contraintes linéaires dans $\mathbb{R}$ ou $\mathbb{Q}$.

Le passage à des domaines discrets ou finis ne simplifie généralement pas la tâche, bien au contraire. Ainsi, le passage de la programmation linéaire à la programmation linéaire en nombres entiers, correspond à un passage de la classe des problèmes polynomiaux à une classe de problèmes que l'on ne sait, pour l'instant, résoudre qu'en temps exponentiel : le problème de décision devient NP-complet.

La contrepartie de cette difficulté est une possible généralité sur la nature des contraintes, car *toute* contrainte sur des variables ayant des domaines finis pourra être exprimée et traitée. D'où des domaines d'application très vastes : interprétation de scènes, raisonnement géométrique, configuration d'équipements, conception, raisonnement temporel, ordonnancement de tâches avec ou sans ressources, élaboration d'emplois du temps, résolution de « puzzles »...

Nous avons déjà examiné quelques techniques de satisfaction en logique propositionnelle au chapitre 6 (algorithme de Quine, procédure de Davis et Putnam, évaluation sémantique). Nous nous consacrerons ici aux problèmes de satisfaction de contraintes quelconques, exprimées *a priori* en extension, dans des domaines *finis* quelconques[4]. En dehors des domaines d'application déjà cités, on notera que certaines des techniques que nous allons présenter sont utilisées comme outil de représentation et de résolution dans des systèmes experts (Berlandier 1988), comme fondement théorique de certains systèmes de maintien de la consistance (Bessière 1991), ou encore et surtout comme technique de base pour la résolution de contraintes dans de nombreux langages de programmation (logique) avec contraintes (cf. §18.7).

14.2 Langage et notations

Définissons rapidement comment peut s'exprimer un CSP dans un langage canonique très pauvre : celui des n-uplets. Le lecteur notera que nous nous permettrons

---

3 L'algorithme du simplexe, conçu en 1947 par G. Dantzig pour résoudre les problèmes d'optimisation d'une fonction linéaire sous contraintes linéaires dans les réels ou les rationnels, a une complexité exponentielle dans le pire des cas. Son succès s'explique par sa relative simplicité et par une complexité *moyenne* qui est polynomiale. D'autres algorithmes, de complexité polynomiale dans le pire des cas, existent. Ils sont moins employés, car souvent moins efficaces en moyenne, trop complexes, ou encore trop mal connus (Karmarkar 1984).

4 Il faut noter que ces limitations (domaines finis, contraintes en extension) ne sont pas *indispensables* pour pouvoir utiliser toutes les techniques décrites dans ce chapitre. Il est généralement élémentaire, pour les algorithmes de satisfaction, de s'appuyer sur des contraintes exprimées par des prédicats. L'emploi des techniques CSP sur des domaines tels que les intervalles de réels (en fait de flottants) a été aussi décrit dans (Davis 1987a; Hyvönen 1989; 1992) par exemple. Ces extensions sont offertes dans certains langages de « programmation par contraintes » s'appuyant sur les techniques CSP tels que PECOS (ILO 1992) ou ECHIDNA (Sidebottom and Havens 1991).

souvent, et sans avertissement, d'utiliser des fonctions ensemblistes sur des n-uplets (ou séquences). On considérera dans ces cas que le n-uplet représente l'ensemble de ses éléments.

Définition 14.1 – *Un problème de satisfaction de contraintes*[5] *ou CSP* $\mathcal{P} = (X, D, C)$ *est défini par :*

- *une séquence* $X = (x_1, \ldots, x_n)$ *de* n *variables ;*
- *une séquence* $D = (d_1, \ldots, d_n)$ *de* n *domaines finis pour les variables de* X*. Le domaine* d_i *est associé à la variable* x_i*.*
- *une séquence* $C = (c_1, \ldots, c_m)$ *de* m *contraintes. Chaque contrainte* c_i *est définie par un couple* (v_i, r_i) *:*
  - v_i *est une séquence de variables* $(x_{i_1}, \ldots, x_{i_{n_i}}) \subset X$ *sur lesquelles porte la contrainte* c_i*. Elle est exprimée en extension. On appelle arité de* c_i *la longueur de la séquence* v_i *;*
  - r_i *est une relation, définie par un sous-ensemble du produit cartésien* $d_{i_1} \times \cdots \times d_{i_{n_i}}$ *des domaines*[6] *associés aux variables de* v_i*. Il représente les n-uplets de valeurs autorisées pour ces variables. Il sera exprimé en extension.*

Définition 14.2 – CSP binaire – *Un CSP binaire est un CSP* (X, D, C) *dont toutes les contraintes* $c_i \in C$ *ont une arité égale à 2.*

Considérons par exemple le CSP suivant, inspiré d'un exemple de (Mackworth 1977) : la table 14.1 définit les ensembles X et D, la table 14.2 définit l'ensemble C *i.e.*, pour chaque contrainte, les variables sur lesquelles elle porte ainsi que sa relation associée.

On peut donner une représentation d'un CSP sous la forme d'un hyper-graphe[7] dont les sommets seront les variables et dont les hyper-arêtes seront les contraintes. Dans le cas de CSP binaires, on pourra se contenter d'un graphe. Il s'agit d'une représentation très usitée dans la littérature, ce qui fait que l'on identifie souvent CSP, hyper-graphe de contraintes, graphe de contraintes (contraintes binaires seulement) et réseau de contraintes[8]. La structure du CSP précédent est ainsi représentée figure 14.1. À l'intérieur de chaque nœud, représentant une variable, on a explicitement indiqué le domaine associé à la variable ; à côté de chaque arc, représentant une contrainte, on a explicitement indiqué la relation qui lui est associée (dans ce cas particulier, on notera que la contrainte entre deux variables connectées peut s'interpréter comme un ordre lexicographique strict, dans le sens des flèches).

---

5 La définition d'un CSP n'est pas très éloignée de celle d'une *instance de base de données* utilisée dans la théorie des bases de données relationnelles (Xuong 1992).

6 Bien souvent, après établissement de la consistance d'arc par exemple (cf. *infra*), un CSP voit ses domaines amputés de quelques éléments et il est alors possible qu'une contrainte $c_i = (v_i, r_i) \in C$ ne vérifie plus $r_i \subset d_{i_1} \times \cdots \times d_{i_{n_i}}$. On considère alors implicitement que la relation associée à c_i est $r_i \cap (d_{i_1} \times \cdots \times d_{i_{n_i}})$.

7 De façon très grossière : un graphe (Berge 1970) est formé de sommets, chaque sommet pouvant être relié à un autre sommet par une arête. Une arête est donc définie par un ensemble de deux sommets. Un graphe est dit complet si toute paire de sommets est reliée par une arête. Un hyper-graphe est formé de sommets, chaque sommet pouvant être relié à *plusieurs* autres sommets par une hyper-arête, définie par un ensemble de sommets dont le cardinal n'est pas imposé. On représente généralement une hyper-arête sous la forme d'un « patatoïde » recouvrant les sommets reliés.

8 Des représentations plus complètes existent : chaque élément de chacun des domaines est représenté par un sommet, une hyper-arête reliant entre eux les sommets participant à un même n-uplet d'une relation. On appelle le graphe ainsi obtenu le micro-graphe du CSP, il définit sa micro-structure.

Variables	Domaines
x_1	$\{a, b, c\}$
x_2	$\{a, b, c\}$
x_3	$\{a, b\}$
x_4	$\{a, b\}$
x_5	$\{a, b\}$

Table 14.1 – Variables et domaines

Contrainte	Variables	Relation
$c_1 = (v_1, r_1)$	$v_1 = (x_3, x_1)$	$r_1 = \{(a, b), (a, c), (b, c)\}$
$c_2 = (v_2, r_2)$	$v_2 = (x_3, x_2)$	$r_2 = \{(a, b), (a, c), (b, c)\}$
$c_3 = (v_3, r_3)$	$v_3 = (x_3, x_4)$	$r_3 = \{(a, b)\}$
$c_4 = (v_4, r_4)$	$v_4 = (x_3, x_5)$	$r_4 = \{(a, b)\}$
$c_5 = (v_5, r_5)$	$v_5 = (x_4, x_5)$	$r_5 = \{(a, b)\}$

Table 14.2 – Contraintes et relations

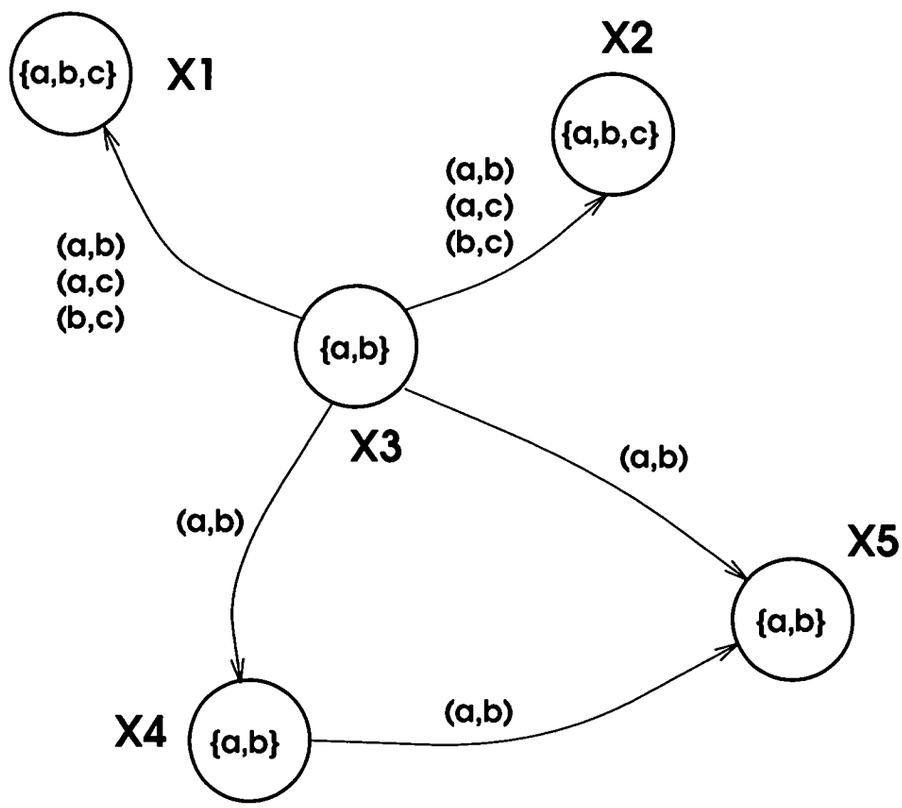


Figure 14.1 – Graphe du CSP

14.3 Sémantique

Tout comme dans le cadre de la logique, il s'agit maintenant de donner un sens précis à un CSP tel qu'il a été défini dans la section précédente.

La notion d'instanciation est le pendant de la notion d'interprétation en logique propositionnelle :

Définition 14.3 – Instanciation – *Étant donné un CSP* $\mathcal{P} = (X, D, C)$*, on appelle instanciation* $\mathscr{A}$ *de* $Y = \{x_{y_1}, \ldots, x_{y_{|Y|}}\} \subset X$ *une application qui associe à chaque variable* $x_{y_i} \in Y$ *une valeur* $\mathscr{A}(x_{y_i}) \in d_{y_i}$ *(le domaine de* x_{y_i}*).*

Dorénavant, nous nous permettrons de faire référence à une instanciation $\mathscr{A}$ aussi bien en tant qu'application, que via le graphe de cette application (l'ensemble des couples $(x_{y_i}, \mathscr{A}(x_{y_i}))$) que nous noterons :

$$\mathscr{A} = \{x_{y_1} \rightarrow \mathscr{A}(x_{y_1}), \ldots, x_{y_{|Y|}} \rightarrow \mathscr{A}(x_{y_{|Y|}})\}$$

ou comme une séquence de valeurs $(\mathscr{A}(x_{y_1}), \ldots, \mathscr{A}(x_{y_{|Y|}}))$, l'ordre des variables de Y étant implicite. On dira d'une instanciation dont le domaine est X qu'elle est complète, elle est dite partielle sinon.

Nous nous permettrons également d'appliquer une instanciation $\mathscr{A}$ à un ensemble ou une séquence V de variables[9]. Le résultat de cette application est alors l'ensemble ou la séquence des images des éléments de V par $\mathscr{A}$.

Définition 14.4 – Satisfaction de contrainte – *Étant donné un CSP* $\mathcal{P} = (X, D, C)$*, une instanciation* $\mathscr{A}$ *de* Y satisfait *la contrainte* $c_i = (v_i, r_i)$ *de* C *(noté* $\mathscr{A} \models c_i$*) ssi* $v_i \subset Y$ *et* $\mathscr{A}(v_i) \in r_i$*. À l'opposé, on dira qu'une instanciation* $\mathscr{A}$ *de* Y viole c_i *ssi* $v_i \subset Y$ *et* $\mathscr{A}(v_i) \notin r_i$*.*

La relation associée à une contrainte définit donc explicitement l'ensemble des valeurs que sont autorisées à prendre simultanément les variables sur lesquelles la contrainte porte. Dans l'exemple précédent, l'affectation :

$$\mathscr{A} = \{x_1 \rightarrow b, x_2 \rightarrow a, x_3 \rightarrow a\}$$

satisfait la contrainte c_1, car le couple $(a, b) = \mathscr{A}((x_3, x_1))$ appartient à r_1, elle viole c_2 car le couple $(a, a) = \mathscr{A}((x_3, x_2))$ n'appartient pas à r_2, enfin, elle ne viole pas et ne satisfait pas la contrainte c_3 car v_3 n'est pas inclus dans $\{x_1, x_2, x_3\}$, l'ensemble des variables instanciées par $\mathscr{A}$.

On définit le *taux de satisfiabilité* d'une contrainte $c_i \in C$ par le rapport entre le cardinal de r_i et le cardinal du produit cartésien des domaines des variables de v_i. Les taux de satisfiabilité de c_1 et de c_2 dans l'exemple précédent sont donc égaux à $\frac{|r_1|}{3\times 2} = \frac{1}{2}$, alors que les taux de satisfiabilité de c_3, c_4 et c_5 sont de $\frac{|r_3|}{2\times 2} = \frac{1}{4}$. Dans certains cas, le taux de satisfiabilité d'une contrainte n'est pas raisonnablement calculable car le calcul de la relation en extension est trop coûteux. Dans d'autre cas, il est immédiat. C'est le cas

9 Comme le permettent les fonctionnelles du type `map` dans les langages fonctionnels.

par exemple des clauses en logique propositionnelle. Une clause non trivialement valide contenant n littéraux a un taux de satisfiabilité de $\frac{2^n-1}{2^n}$.

Définition 14.5 – Instanciation consistante – *Étant donné un CSP* $\mathcal{P} = (X, D, C)$, *une instanciation* $\mathcal{A}$ *des variables de* $Y \subset X$ *est dite consistante*[10] *ssi :*

$$\forall c_i = (v_i, r_i) \in C \text{ telle que } v_i \subset Y, \mathcal{A} \models c_i$$

Autrement dit, une instanciation est consistante si elle ne viole aucune contrainte. Vérifier qu'une instanciation sur Y est consistante est un problème facile (polynomial). Il suffit de vérifier que chaque contrainte c_i n'est pas violée par cette instanciation.

Définition 14.6 – Solution(s) d'un CSP – *Une solution*[11] $\mathcal{S}$ *de* $\mathcal{P} = (X, D, C)$ *est une instanciation consistante des variables de* X. *On dit alors que l'instanciation* $\mathcal{S}$ *satisfait* $\mathcal{P}$ *(noté* $\mathcal{S} \models \mathcal{P}$*). L'ensemble des solutions de* $\mathcal{P}$ *sera noté* $S_{\mathcal{P}}$.

On peut considérer qu'un CSP $\mathcal{P}$ exprime en fait, sous une forme généralement plus concise, l'ensemble de ses solutions ou plus précisément qu'il définit une contrainte sur X, dont la relation associée est tout simplement $S_{\mathcal{P}}$. Vérifier qu'une instanciation complète est une solution est un problème facile.

Définition 14.7 – Consistance d'un CSP – *Un CSP* $\mathcal{P} = (X, D, C)$ *est dit* consistant *ssi* $S_{\mathcal{P}} \neq \emptyset$.

Le problème de décision « Un CSP est-il consistant ? » est NP-complet[12] (cf. chapitre 11). On peut définir une propriété sur les instanciations plus forte que la consistance et dont la vérification est aussi difficile (*i.e.*, elle définit elle aussi un problème de décision NP-complet) :

Définition 14.8 – Instanciation globalement consistante – *Étant donné un CSP* $\mathcal{P} = (X, D, C)$, *une instanciation* $\mathcal{A}$ *de* $Y \subset X$ *est dite* globalement consistante *ssi* $\exists \mathcal{S} \in S_{\mathcal{P}}$ *telle que* $\mathcal{A} \subset \mathcal{S}$.

Si une instanciation n'est pas globalement consistante, il n'est pas possible de l'étendre en une solution. Cette propriété est d'autant plus difficile à vérifier (et donc d'autant plus intéressante à connaître) que l'instanciation est de petite taille. On note en particulier que la consistance globale de l'instanciation vide est équivalente à la consistance du CSP. Le lecteur pourra constater que le CSP de la figure 14.1 n'a pas de solution (il est dit inconsistant).

Afin d'illustrer les définitions précédentes, et pour mieux cerner le sens d'un CSP, considérons l'exemple suivant, présenté dans (Bessière 1992b), et sa formalisation sous la forme d'un CSP :

---

10 Ou localement consistante, par opposition à globalement consistante (cf. *infra*). Les auteurs français emploient indifféremment les mots de cohérence et de consistance, ce dernier étant dérivé de l'anglais *consistency*. Nous avons employé le dernier par souci de cohérence avec les chapitres consacrés à la logique mathématique.

11 À rapprocher de la notion de modèle en logique propositionnelle.

12 On peut le prouver simplement par restriction. Le problème NP-complet de k-colorabilité d'un graphe s'énonce naturellement sous la forme d'un CSP dont les variables sont les sommets du graphe, les domaines sont les colorations possibles, une contrainte étant associée à chaque arête du graphe et exprimant que deux sommets adjacents ne doivent pas être colorés de la même façon. L'appartenance à NP découle simplement de nos remarques sur la polynomialité de la vérification qu'une instanciation est solution.

La firme Peunault va sortir un nouveau modèle de voiture fabriqué dans toute l'Europe :

- *les* portières *et le* capot *sont faits à Lille, où le constructeur ne dispose que de peinture* rose, rouge *et* noire ;
- *la* carrosserie *est faite à Hambourg, où l'on a de la peinture* blanche, rose, rouge *et* noire ;
- *les* pare-chocs, *faits à Palerme, sont toujours* blancs ;
- *la bâche du* toit-ouvrant, *qui est faite à Madrid, ne peut être que* rouge ;
- *les* enjoliveurs *sont faits à Athènes, où l'on a de la peinture* rose *et de la peinture* rouge.

Le concepteur de la voiture impose quelques-uns de ses désirs quant à l'agencement des couleurs pour cette voiture :

- *la* carrosserie *doit être de la même couleur que les* portières, *les* portières *de la même couleur que le* capot, *le* capot *de la même couleur que la* carrosserie[13] ;
- *les* enjoliveurs, *les* pare-chocs *et le* toit-ouvrant *doivent être plus clairs que la* carrosserie.

Ce problème peut se coder sous la forme d'un CSP binaire (toutes les contraintes portent sur au plus deux variables) dont les variables $x_1, x_2, x_3, x_4, x_5, x_6$ correspondent aux différents composants de la voiture, les couleurs de peinture *blanc, rose, rouge, noir* sont représentées par les symboles a, b, c, d. Les contraintes et leurs relations découlent simplement du problème. Le graphe du CSP est donné figure 14.2 (dans ce cas particulier, on notera qu'un arc a été utilisé pour les contraintes dont la relation associée peut s'interpréter comme une relation d'ordre, et une arête pour les contraintes dont la relation associée est une relation d'égalité). Ce problème simple possède deux solutions seulement, il s'agit des instanciations :

$$\mathscr{S}_1 = \{x_1 \rightarrow a, x_2 \rightarrow d, x_3 \rightarrow d, x_4 \rightarrow b, x_5 \rightarrow d, x_6 \rightarrow c\}$$
$$\mathscr{S}_2 = \{x_1 \rightarrow a, x_2 \rightarrow d, x_3 \rightarrow d, x_4 \rightarrow c, x_5 \rightarrow d, x_6 \rightarrow c\}$$

On en déduit donc que l'instanciation $\{x_2 \rightarrow d, x_3 \rightarrow d, x_6 \rightarrow c\}$ par exemple est globalement consistante, alors que l'instanciation *consistante* $\{x_1 \rightarrow a, x_3 \rightarrow c, x_4 \rightarrow b\}$ ne l'est pas.

Définition 14.9 – Équivalence de CSP – *Les CSP* $\mathcal{P} = (X, D, C)$ *et* $\mathcal{P}' = (X, D', C')$ *sont dits équivalents*[14] *(noté* $\mathcal{P} \equiv \mathcal{P}'$*) ssi* $S_{\mathcal{P}} = S_{\mathcal{P}'}$.

Si nous disposons de deux CSP équivalents, nous pouvons décider de résoudre l'un à la place de l'autre : ils ont le même ensemble de solutions. Ainsi, considérons le CSP $\mathcal{P}$ de la figure 14.2 et le CSP $\mathcal{P}'$ obtenu à partir de $\mathcal{P}$, en supprimant de d_3 (le domaine de x_3) les valeur b et c. Ces deux CSP ont le même ensemble de solutions et sont donc

---

13 Cette dernière contrainte est évidemment inutile car elle est *induite* (cf. *infra*) par les deux précédentes. S'il est nécessaire, par la suite, de faire référence aux désirs du concepteur, il est indispensable de modéliser *toutes* les contraintes exprimées, même si il y a redondance.

14 Une relation d'équivalence plus faible est proposée dans (Rossi *et al.* 1990). Celle que nous donnons ici est la plus communément admise.

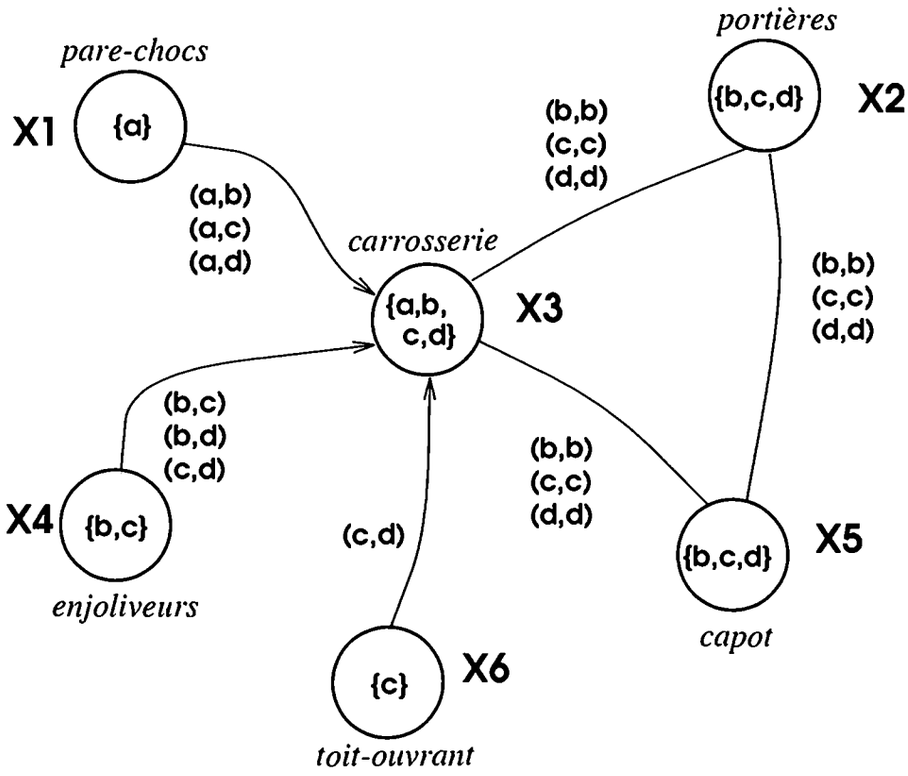


Figure 14.2 – Un CSP pour la « conception » de voitures décoratives ?

équivalents, mais $\mathcal{P}'$ est intuitivement plus simple que $\mathcal{P}$, le nombre d'instanciations complètes possibles étant plus faible.

Définition 14.10 – Contrainte induite – *Étant donnés un CSP* $\mathcal{P} = (X, D, C)$ *et une contrainte* c, *définie par un ensemble* $v_c \subset X$ *de variables qu'elle connecte et une relation* r_c, *on dira que* c *est induite*[15] *par* $\mathcal{P}$ *ssi* $\forall \mathscr{S} \in S_{\mathcal{P}}, \mathscr{S} \models c$.

Autrement dit, une contrainte c (en général, ce n'est pas une contrainte de C) est induite par un CSP $\mathcal{P}$ si elle est satisfaite par *toutes* les solutions de $\mathcal{P}$. Cette notion est intimement liée à la notion d'instanciation globalement consistante : une contrainte est induite par un CSP ssi seuls des n-uplets représentant une instanciation non globalement consistante sont absents de sa relation associée.

Une contrainte induite peut, et c'est là tout son intérêt, être rajoutée au CSP qui l'induit sans modifier l'ensemble de ses solutions ! Ainsi, et dans le cadre du CSP $\mathcal{P}$ illustré figure 14.2, la contrainte c portant sur l'unique variable x_3 et dont la relation associée est définie par l'ensemble $\{(a), (d)\}$ est une contrainte induite du CSP. On peut donc l'ajouter à $\mathcal{P}$ pour obtenir un CSP qui n'est finalement pas très différent du CSP $\mathcal{P}'$ obtenu précédemment par suppression des valeurs b et c du domaine de x_3 (de façon équivalente, on notera que les deux instanciations $\mathscr{A}_1 = \{x_3 \rightarrow b\}$ et $\mathscr{A}_2 = \{x_3 \rightarrow c\}$ ne sont pas globalement consistantes).

Définition 14.11 – Consistance globale – *Un CSP* $\mathcal{P} = (X, D, C)$ *est dit* globalement consistant *ssi toute instanciation consistante est globalement consistante.*

Cette propriété est extrêmement intéressante puisqu'alors, toute instanciation *partielle* consistante est globalement consistante et pourra donc être étendue de façon glou-

15 Cette relation pourrait aussi se définir entre CSP, ou entre contraintes. Elle est le pendant de la notion de conséquence valide en logique propositionnelle.

tonne en une solution. La construction d'une solution d'un CSP globalement consistant est donc un problème trivial. Aucun des CSP que nous avons considéré jusqu'ici n'est globalement consistant.

Définition 14.12 – Minimalité – *Un CSP* $\mathcal{P} = (X, D, C)$ *est dit* minimal *ssi :*

$$\forall c_i = (v_i, r_i) \in C, \bigcup_{\mathscr{S} \in S_{\mathcal{P}}} \left[\mathscr{S}(v_i)\right] = r_i$$

Habituellement, la minimalité suppose en plus que le CSP est binaire (Montanari 1974).

Autrement dit, un CSP est minimal si toutes ses contraintes sont les contraintes minimales (au sens de l'inclusion sur les relations associées) induites par lui-même. Un CSP binaire globalement consistant est nécessairement minimal. On dit alors qu'il est décomposable (la relation n-aire exprimée par le CSP est décomposable en relations binaires). Le contraire est bien sûr faux dans le cas général. Si l'existence d'une solution est immédiate à établir pour un CSP minimal, la difficulté de construction d'une solution est un problème qui, à ma connaissance, reste ouvert.

14.4 Énoncés de problèmes

Un CSP étant donné, de nombreux problèmes peuvent être considérés. Le problème habituellement considéré (et d'où les CSP tirent leur nom) est le problème de *satisfaction* qui consiste à exhiber une solution du CSP. Ce problème est, dans le cas général, NP-difficile *i.e.*, au moins aussi difficile qu'un problème NP-complet (cf. chapitre 11).

De nombreux autres problèmes peuvent être associés à un CSP :

- prouver qu'un CSP est consistant, plus connu sous le nom de satisfiabilité. Il s'agit d'un problème NP-complet ;
- exhiber toutes les solutions d'un CSP ;
- exhiber une solution d'un CSP qui maximise (strictement ou « raisonnablement »[16]) un ou plusieurs critères (éventuellement antagonistes) ;
- calculer ou estimer le nombre de solutions d'un CSP ;
- trouver pour une (ou plusieurs) variable(s), un ensemble de valeurs qui figurent dans toutes les solutions...

14.4.1 Algorithme standard pour la satisfaction

Pour résoudre le problème de satisfaction, les méthodes exposées dans le chapitre 13 conviennent. On considère un espace où chaque état est une instanciation $\mathscr{A}$, les règles de production permettant d'étendre l'instanciation $\mathscr{A}$ sur une variable avec toutes les valeurs possibles de son domaine. Les feuilles de l'arbre ainsi exploré sont des instanciations complètes dont on pourra aisément vérifier si elles sont, ou non, des solutions du CSP traité. Le parcours se réalise toujours en profondeur d'abord : un parcours en *largeur d'abord* est inintéressant puisque les solutions sont à une profondeur connue.

16 *i.e.*, une solution qui est proche de l'optimum sans que l'on puisse garantir que cet optimum a été atteint.

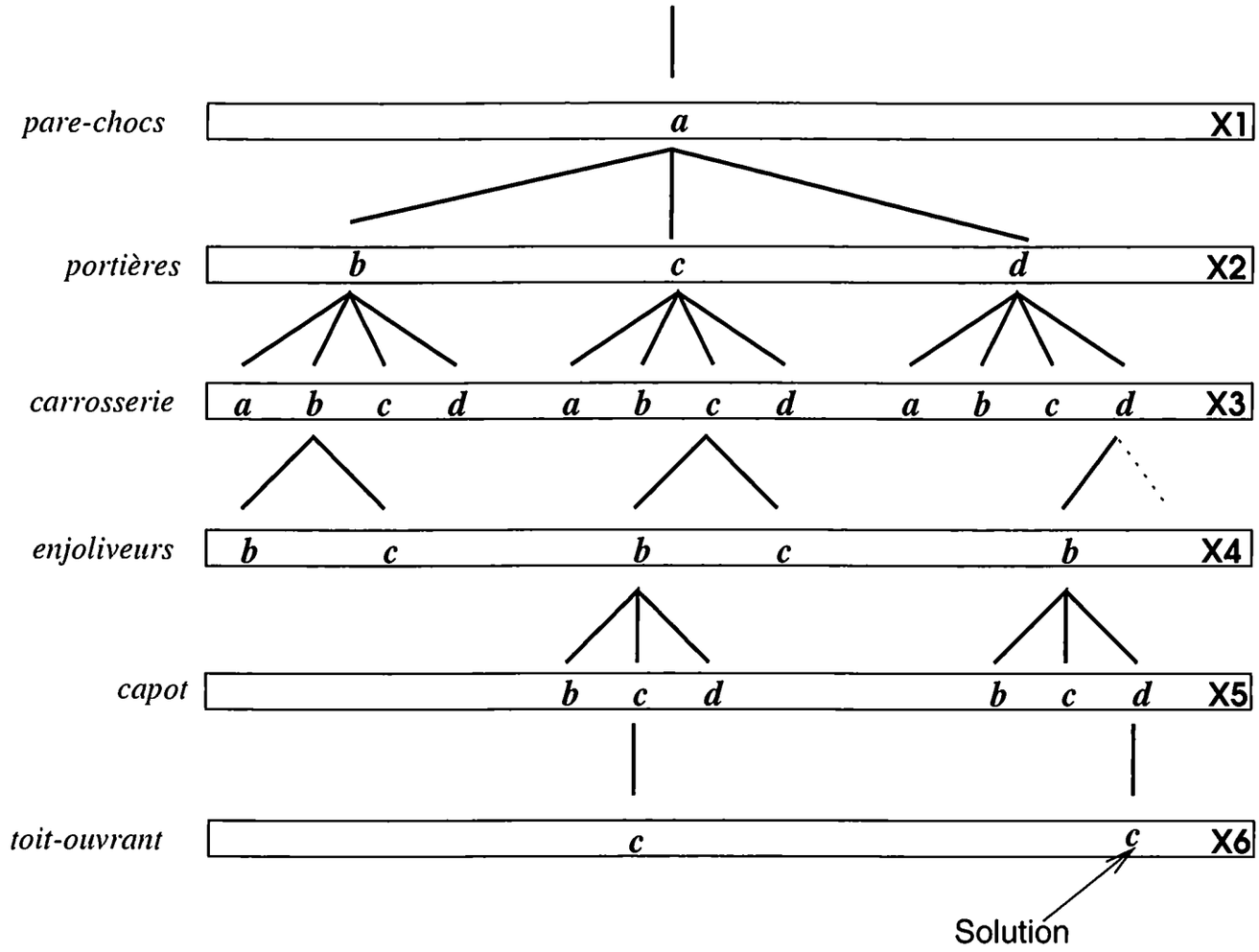


Figure 14.3 – Application de l'algorithme Backtrack

L'algorithme obtenu est désigné sous le nom de *generate and test*. Sur un CSP de n variables, avec des domaines de cardinal d et m contraintes, l'ensemble des états terminaux a un cardinal de d^n, et on peut être amené, pour obtenir toutes les solutions, à évaluer $m * d^n$ contraintes[17].

Il suffit en fait qu'une contrainte soit violée pour qu'aucune des instanciations filles d'un nœud ne puisse être consistante. Or, il est possible de vérifier qu'une contrainte est violée dès que toutes les variables sur lesquelles elle porte ont été instanciées. Cette propriété peut se résumer simplement par :

Propriété 14.1 – *Toute affectation non consistante est non globalement consistante.*

Une amélioration triviale consistant à vérifier la satisfaction de chaque contrainte dès que possible nous conduit donc à l'algorithme de recherche *test and generate*, souvent référencé sous le nom de backtrack ou *BT*. Lors d'un parcours en profondeur d'abord, l'arrivée sur un nœud terminal entraîne un retour-arrière sur le nœud père. L'algorithme est incarné dans la fonction récursive backtrack qui prend en argument une séquence V de variables à instancier (initialement X en entier) et une instanciation $\mathscr{A}$ (initialement vide). Elle est décrite informellement[18] comme l'algorithme 1.

---

17 À titre indicatif, s'il faut 10^{-6}s pour évaluer une contrainte, le traitement d'un CSP de 20 variables, avec des domaines de cardinal 5 et avec 20 contraintes ne prendrait pas moins de 60 années...

18 Pour des raisons de simplicité d'exposition, les algorithmes sont présentés sous une forme simple, sans rentrer dans les détails de la réalisation. La majorité des algorithmes de satisfaction présentés dans ce chapitre sont généralement détaillés dans les publications correspondantes.

Algorithme 1: Backtrack

```
backtrack(V, 𝒜);
si V = ∅ alors
    𝒜 est une solution;
sinon
    soit x_i ∈ V;
    pour chaque ν ∈ d_i faire
        si 𝒜 ∪ {x_i → ν} est consistante alors
            backtrack(V − {x_i}, 𝒜 ∪ {x_i → ν});
```

La figure 14.3 donne l'arbre de recherche exploré par l'algorithme appliqué au problème de conception de la figure 14.2 et en instanciant les variables dans l'ordre $(x_1, x_2, x_3, x_4, x_5, x_6)$. Il faut explorer 29 nœuds et effectuer 30 vérifications de contraintes avant de trouver la première solution.

L'algorithme backtrack souffre de nombreux défauts : parcours aveugle de l'espace sans prise en compte d'informations « évidentes » sur la non consistance globale d'instanciations partielles, découverte répétée des mêmes inconsistances locales...

Plusieurs approches ont permis d'aboutir à des algorithmes de traitement beaucoup plus efficaces :

1. définition de problèmes polynomiaux qui correspondent à des versions plus ou moins « affaiblies » de la propriété de satisfiabilité. Ont été ainsi définies les propriétés de consistance locale (cf. §14.5) ;
2. améliorations de l'efficacité de l'algorithme backtrack. Plusieurs types de sophistications ont vu le jour (cf. §14.6) ;
3. définition de méthodes de décomposition de CSP en sous-CSP plus simples, si possible dont la satisfiabilité soit un problème polynomial (cf. §14.7).

Ce sont ces trois approches que nous allons maintenant présenter.

14.5 Consistances locales et filtrage

Le « simple » problème de satisfiabilité d'un CSP étant NP-complet, on a été amené à définir des propriétés moins fortes que la consistance. Très grossièrement, l'idée est de limiter la propriété de consistance à des « sous-CSP » de taille k. De nombreuses propriétés de *consistance locale* ont été ainsi définies dans (Montanari 1974; Mackworth 1977) par exemple. Toutes ces propriétés peuvent être généralisées (Freuder 1978) :

Définition 14.13 – k-consistance – *Un CSP est dit k-consistant si pour tout n-uplet de k variables $(x_1, \ldots, x_k)$, pour toute instanciation $\mathscr{A}$ consistante des $(k-1)$ variables $(x_1, \ldots, x_{k-1})$, il existe une valeur $\nu \in d_k$ telle que l'instanciation $\mathscr{A} \cup \{x_k \rightarrow \nu\}$ soit consistante.*

Informellement, un CSP est k-consistant ssi toute instanciation consistante de $(k-1)$ variables peut être étendue à toute k-ième variable pour constituer une instan-

ciation de k variables qui soit consistante. Cette propriété ne peut être, seule, directement liée à la consistance du CSP. On définit donc une propriété plus forte.

Définition 14.14 – k-consistance forte – *Un CSP $\mathcal{P}$ est dit fortement k-consistant ssi, $\forall i, 1 \leq i \leq k$, $\mathcal{P}$ est i-consistant.*

Un CSP comprenant n variables qui est fortement n-consistant est globalement consistant. Il faut cependant se rappeler que le fait qu'une propriété de consistance locale donnée soit établie ne permet pas, dans le cas général, d'assurer la consistance.

14.5.1 Filtrages

On peut ainsi poser le problème d'*établissement de consistance locale* ou *filtrage* en termes plus précis : étant donné un CSP $\mathcal{P} = (X, D, C)$, construire un CSP $\mathcal{P}' = (X, D', C')$ qui soit équivalent à $\mathcal{P}$ et qui soit k-consistant. Le CSP $\mathcal{P}'$ présente alors l'avantage d'être bien souvent plus simple à résoudre par un algorithme de recherche du type backtrack.

Tous les algorithmes de filtrage fonctionnent suivant un même principe : on va localement et massivement inférer des contraintes *induites* par le CSP et les ajouter au CSP jusqu'à ce qu'il ne soit plus possible d'inférer de contraintes qui ne soient déjà explicites dans le CSP[19]. Tous ces algorithmes possèdent des traits d'un intérêt indéniable :

1. ils peuvent être interrompus à tout instant pour fournir un CSP qui n'est pas encore k-consistant mais qui est tout de même plus simple que le CSP initial ;
2. toute contrainte induite par un CSP (X, D, C) est aussi induite par un CSP (X, D, C') si $C \subset C'$. Ces algorithmes sont donc naturellement incrémentaux pour l'ajout de contraintes. La suppression de contraintes est beaucoup plus problématique (cf. §14.8).

Un algorithme général, permettant d'établir la forte k-consistance a été proposé dans (Cooper 1989). Appliqué à un CSP de n variables dont les domaines sont de cardinal d, sa complexité spatiale et temporelle est en $O(n^k.d^k)$ (notation de Landau, cf. (Beauquier *et al.* 1992, § 2.1)). De ce fait, les algorithmes de filtrage les plus utilisés sont ceux qui établissent une consistance de faible niveau.

14.5.2 Arc-consistance

Il s'agit de la plus ancienne et de la plus utilisée des consistances locales (Ullman 1966; Waltz 1972; Montanari 1974; Mackworth 1977). Elle correspond, dans le cas des CSP binaires, à la 2-consistance forte.

L'idée est simple : si une valeur ν du domaine de x_i n'apparaît dans aucun des n-uplets d'une contrainte portant sur x_i, il est certain que ν ne participera pas à une solution. On dit d'une telle valeur qu'elle n'a pas de support sur les autres variables de la contrainte ou encore qu'elle n'est pas viable. On dira qu'un CSP est arc-consistant s'il ne contient pas de valeurs non viables. Plus formellement :

---

19 L'arrêt est garanti et définit un CSP unique dans tous les cas qui nous intéressent. L'ajout de contraintes qui sont *induites* garantit d'autre part l'équivalence des CSP.

Définition 14.15 – Arc-consistance – *Un CSP* $\mathcal{P} = (X, D, C)$ *est arc-consistant ssi pour toute variable* $x_i \in X$, *on a* $d_i \neq \varnothing$ *et* $\forall v \in d_i, \forall c_j = (v_j, r_j) \in C \mid x_i \in v_j$, *il existe* $\mathscr{A} \in r_j$ *tel que* $\mathscr{A}(x_i) = v$.

De nombreuses procédures d'établissement de la consistance d'arc ont été exhibées. Nous ne considérerons dans la suite de cette section que les CSP binaires[20]. On se limite ici au calcul de contraintes *unaires*, induites par des « sous-CSP » définis par des couples de variables. Les algorithmes les plus anciens s'appuient sur une procédure très simple, appelée Revise, qui s'applique à une paire de variables (x_i, x_j) reliées par une contrainte c_k. Revise supprime du domaine de x_i toute valeur qui rend impossible la satisfaction de la contrainte entre x_i et x_j et retourne, en sus des effets de bord sur les domaines, un booléen qui indique si le domaine de x_i a été modifié.

Algorithme 2: Revise

```
Revise(x_i, x_j);
soit modification ← faux;
pour chaque v ∈ d_i faire
    si ∄v' ∈ d_j | {x_i → v, x_j → v'} est consistante alors
        d_i ← (d_i − {v});
        modification ← vrai;
retourner modification;
```

Cette procédure est appliquée sur tous les couples de variables. Le problème essentiel est que la réduction du domaine d_i peut entraîner l'apparition de nouvelles valeurs arc-inconsistantes sur d'autres variables reliées par une contrainte à x_i et qu'une seule itération ne suffit généralement pas. Dans la version la plus primitive appelée AC1, on répète l'application sur tous les couples de variables (dans les deux sens) et jusqu'à ce que la procédure Revise retourne *faux* pour tous les couples de variables. Dans la version appelée AC3, une file L est utilisée pour éviter l'application inutilement répétée de Revise. Si l'on considère un CSP formé de m contraintes et dont les domaines ont un cardinal de d, on obtient ainsi une complexité en $O(m.d^3)$.

Algorithme 3: AC-3

```
AC-3();
soit L ← liste des paires (x_i, x_j) | ∃ une contrainte entre x_i et x_j;
tant que L ≠ ∅ faire
    Choisir et supprimer dans L une paire (x_i, x_j);
    si Revise(x_i, x_j) alors
        L ← L ∪ {(x_k, x_i) | ∃ une contrainte entre x_k et x_i};
```

La première version de complexité optimale dans le pire des cas, appelée AC-4 (Mohr and Henderson 1986), ne s'appuie plus sur la procédure Revise. L'algorithme AC-4 détermine dans une première phase :

20 Un certain nombre d'algorithmes ont été étendus aux contraintes n-aires (Mohr and Masini 1988; Janssen 1990; Jégou 1991; Bessière and Régin 1997).

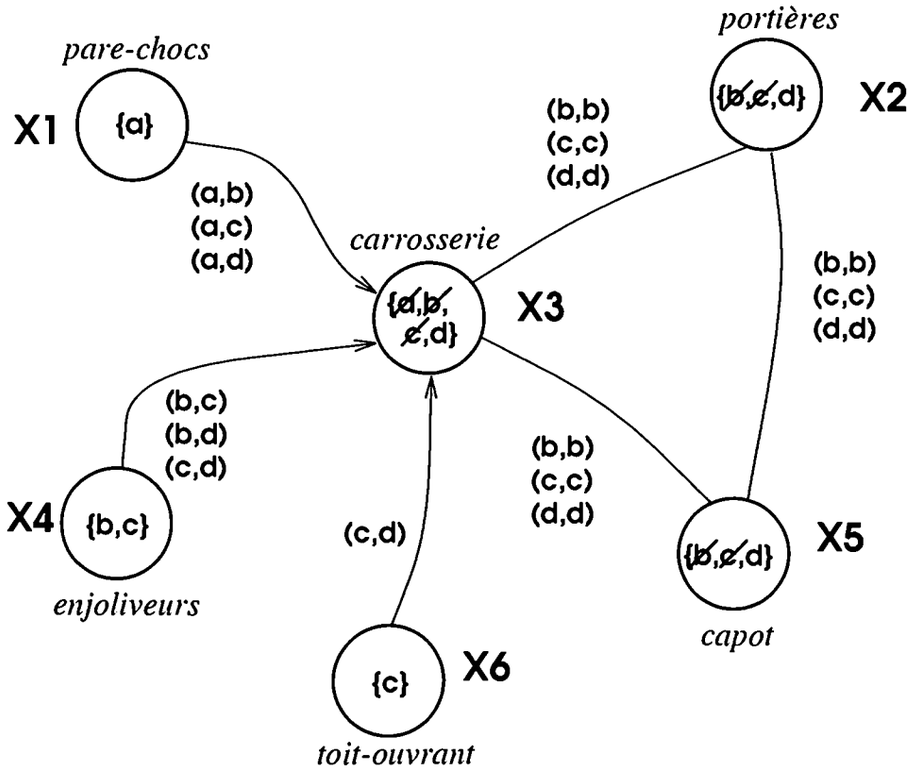


Figure 14.4 – Le CSP après établissement de l'arc-consistance.

1. pour toute paire de variables (x_i, x_j) reliées par une contrainte, et pour tout élément ν de d_i, le nombre d'éléments du domaine d_j de x_j qui sont compatibles avec ν (noté $cpt[(i,j),\nu]$) ;
2. pour tout élément μ de d_j, l'ensemble $S_{j,\mu}$ des paires (i,ν) telles que x_i et x_j sont reliées par une contrainte et ν est compatible avec μ.

Enfin, chaque valeur $\nu \in d_i$ dont un des compteurs $cpt[(i,j),\nu]$ est nul est mise en évidence en positionnant une variable booléenne $mark(i,\nu)$, initialement nulle, à 1.

Ces informations sont conservées pour la deuxième phase de l'algorithme : les valeurs marquées, qui ont un compteur $cpt[(i,j),\nu]$ nul, ont été supprimées car il n'y a plus de valeur de d_j qui soit compatible avec ν. On doit alors décrémenter les compteurs de toutes les valeurs encore présentes qui étaient compatibles avec ν et qui sont référencées dans $S_{i,\nu}$. Il se peut alors que de nouveaux compteurs atteignent 0...

La complexité temporelle obtenue est celle de la première phase, en $O(m.d^2)$ (donc linéaire dans la taille des données : il y a m contraintes, chacune contenant de l'ordre de d^2 n-uplets). Elle présente l'inconvénient, d'une complexité spatiale importante. Une généralisation aux CSP non nécessairement binaires est donnée dans (Mohr and Masini 1988). Une version générique, appelée AC5, spécialisable à certains types de contraintes, permet d'obtenir dans certains cas des complexités en $O(m.d)$ (cf. (van Hentenryck *et al.* 1992; Perlin 1992)).

Une version plus récente, appelée AC6 et présentée dans (Bessière and Cordier 1993), permet un gain notable par rapport à ces versions. Ce sont les structures de données d'AC4 qui sont à l'origine de sa complexité optimale dans le pire des cas. Mais la mémorisation et le dénombrement des valeurs compatibles avec chaque valeur, pour chaque contrainte, sont aussi la source de sa complexité spatiale importante et d'une complexité temporelle *moyenne* qui augmente avec le nombre de paires autorisées dans les relations

Algorithme 4: Initialisation AC-4

```
Initialisation-AC-4();
soit L ← liste des paires (x_i, x_j) | ∃ une contrainte entre x_i et x_j;
tant que L ≠ ∅ faire
    Choisir et supprimer dans L une paire (x_i, x_j);
    pour chaque ν ∈ d_i faire
        cpt[(i, j), ν] ← 0;
        pour chaque μ ∈ d_j faire
            si {x_i → ν, x_j → μ} est consistante alors
                cpt[(i, j), ν] ← cpt[(i, j), ν] + 1;
                S_{j,μ} ← S_{j,μ} ∪ {(i, ν)};
        si cpt[(i, j), ν] = 0 alors
            d_i ← (d_i − {ν});
            mark(i, ν) ← 1;
```

Algorithme 5: Propagation AC-4

```
Propagation-AC-4();
soit L ← liste des paires (i, ν) telles que mark(i, ν) = 1;
tant que L ≠ ∅ faire
    Choisir et supprimer dans L une paire (i, ν);
    pour chaque (j, μ) ∈ S_{i,ν} faire
        si mark(j, μ) = 0 alors
            cpt[(j, i), μ] ← cpt[(j, i), μ] − 1;
            si cpt[(j, i), μ] = 0 alors
                L ← L ∪ {(j, μ)};
                mark(j, μ) ← 1;
                d_j ← (d_j − {μ});
```

(puisque le nombre de paires compatibles est directement lié au nombre de paires dans les relations).

AC6 s'appuie sur les mêmes principes qu'AC4, mais au lieu de tester et de dénombrer, pour chaque contrainte, *toutes les valeurs compatibles* avec une valeur $\nu \in d_i$, il s'arrête, pour chaque contrainte, dès l'obtention d'une première valeur compatible qui sera seule mémorisée. Si une valeur qui est la première valeur compatible avec certaines valeurs vient à être effacée, AC6 cherchera une nouvelle valeur compatible pour chacune de ces valeurs. On aboutit ainsi à une complexité spatiale en $O(m.d)$ seulement, à une complexité temporelle bien meilleure en moyenne, et ce sans dégrader la complexité dans le pire des cas. Cet algorithme a été sophistiqué pour aboutir aux algorithmes AC7 et AC-Inférence, présentés dans (Bessière *et al.* 1995) :

- AC7 exploite la propriété dite de bidirectionalité des contraintes i.e., le fait que si la valeur ν de la variable x_i est compatible avec la valeur μ de x_j alors l'inverse est vrai. Il a les mêmes complexités temporelles et spatiales qu'AC6 dans le pire des cas

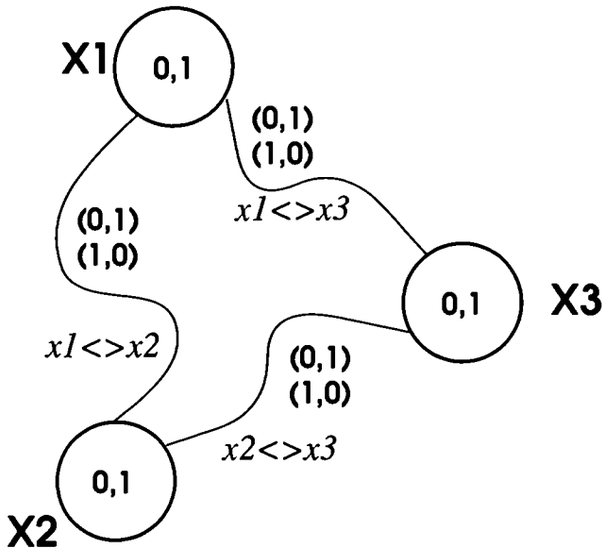


Figure 14.5 – Un CSP arc-consistant et inconsistant

mais peut être sensiblement plus efficace qu'AC6 en moyenne. AC7 a servi de base au développement d'une version paresseuse, plus efficace quand seule la détection d'incohérence est utile, est donnée dans (Schiex *et al.* 1996) et a été également étendu aux CSP non binaires dans (Bessière and Régin 1997).

- AC-inférence est une extension d'AC7 qui peut exploiter les propriétés de certaines contraintes telles que la réflexivité, la commutativité... La complexité spatiale dans le pire des cas est alors en $O(e.d^2)$.

Après ces étapes de sophistication des algorithmes, ces dernières années ont permis de réconcilier la simplicité du schéma de propagation d'AC3 avec les complexités optimales d'AC-4, AC-6 et AC-7 en aboutissant aux algorithmes AC2000 et AC2001 (Bessière and Régin 2001).

La figure 14.4 montre le CSP obtenu après application d'une procédure d'établissement de l'arc-consistance au CSP de la figure 14.2. Dans ce cas précis, le CSP obtenu est *globalement* consistant. La recherche d'une solution via l'algorithme backtrack est alors gloutonne : elle s'effectue sans que l'on ait jamais à effectuer de retour-arrière. Rappelons qu'il ne s'agit pas d'un cas général : la figure 14.5 montre un exemple de CSP arc-consistant mais inconsistant.

Malgré le faible coût d'établissement de l'arc-consistance, on a défini un certain nombre d'algorithmes qui établissent un sous-ensemble de l'arc-consistance. Comme nous le verrons, ils sont surtout utilisés en collaboration avec l'algorithme backtrack. Si l'on considère que les n variables d'un CSP sont ordonnées, on peut définir des applications restreintes de la procédure Revise à partir d'une variable x_i donnée :

- *check backward :* on applique la procédure Revise à tous les couples (x_i, x_k) tels que $1 \leq k < i$;
- *check forward :* on applique la procédure Revise à tous les couples (x_k, x_i) tels que $i < k \leq n$;
- *partial look future :* on applique la procédure Revise à tous les couples (x_j, x_k) tels que $i \leq j < k \leq n$;
- *full look future :* on applique la procédure Revise à tous les couples (x_j, x_k) tels que $i \leq j \neq k \leq n$;

Ces différentes restrictions sont illustrées figure 14.6.

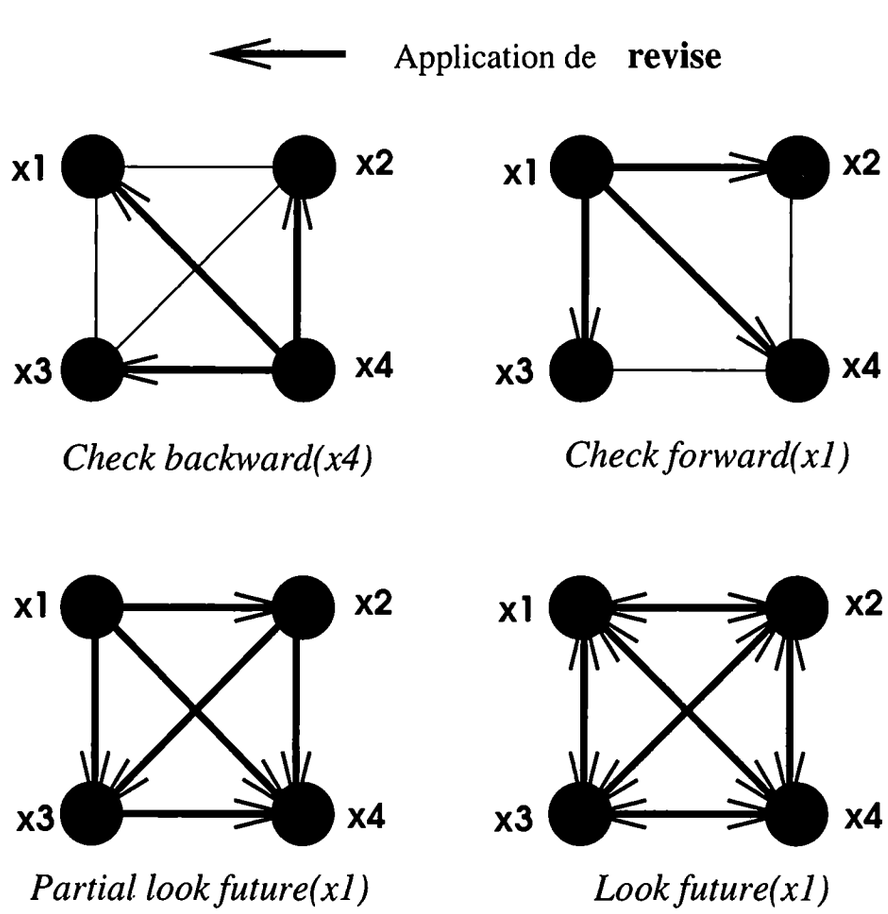


Figure 14.6 – *Arc-consistances* dégradées

14.5.3 Chemin-consistance

La chemin-consistance correspond, dans le cas de CSP binaires, à la 3-consistance forte. Malgré une complexité spatiale et temporelle assez lourde, elle reste intéressante dans le cas de certains types de problèmes pour lesquels la 3-consistance forte assure la consistance globale (cf. §14.7).

La procédure de filtrage correspondante est assez délicate à mettre en œuvre car son application entraîne une modification des relations associées aux contraintes, voire la création de nouvelles contraintes[21].

Bien que la complexité reste polynomiale, la chemin-consistance est expérimentalement jugée trop lourde pour aider un algorithme d'exploration de façon efficace[22]. Les algorithmes les plus élaborés (Mohr and Henderson 1986; Han and Lee 1988; Singh 1995; Assef Chmeiss 1996) ont une complexité temporelle en $O(n^3d^3)$, et une complexité spatiale en $O(n^3d^2)$. Il permettent d'obtenir à la fois la chemin-consistance et l'arc-consistance.

14.5.4 Autres propriétés de consistances locales

De nombreuses autres propriétés de consistance locale ont été définies. Citons :

---

21 Il est relativement aisé de la mettre en œuvre sur une représentation des contraintes sous forme d'ensemble de n-uplets, mais une représentation sous forme de prédicats rend la chose plus délicate.

22 Il s'agit d'une constatation expérimentale et en moyenne. Le choix du niveau de consistance locale à établir avant (ou pendant) une exploration arborescente reste purement heuristique (Haralick and Elliot 1980).

- la *paire-consistance* (Janssen *et al.* 1989), issue des bases de données et qui n'entre pas dans la classe des k-consistances. Elle est particulièrement intéressante dans le cas de CSP d'arité importante (et plus spécialement dans le cadre d'ensembles de clauses de la logique propositionnelle) ;
- l'*arc-consistance* et la *chemin-consistance directionnelles* (introduites dans (Dechter and Pearl 1988a)). Elles limitent les propriétés de consistance locale au plus juste quand le filtrage est utilisé préalablement à une recherche de solution via l'algorithme backtrack et en utilisant un ordre vertical statique (cf. *infra*) ;
- la (i, j)-*consistance* a été introduite par Freuder (Freuder 1985; 1988) comme généralisation de la k-consistance. Elle reste jusqu'ici d'un intérêt essentiellement théorique ;
- La consistance de chemin réduite (*Restricted Patch Consistency* (Berlandier 1995)) et ses derivées (Debruyne and Bessière 1997a; 1997b) (k-Restricted Patch Consistency et Max-Restricted Patch Consistency ainsi que d'autres propositions exotiques) tentent de fournir une propagation plus forte que l'arc-consistance à un coût spatial et temporel raisonnable.
- la *consistance adaptative*, introduite dans (Dechter and Pearl 1988a), est présentée comme une consistance directionnelle, dont l'établissement se fait via un algorithme de complexité exponentielle. Il s'agit en fait d'une variante améliorée du *tree-clustering*. Elle garantit une recherche de solution via backtrack *sans retour-arrière*. Les expérimentations très limitées effectuées par les auteurs semblent montrer qu'elle est moins intéressante que l'utilisation d'un simple algorithme du type **graph based backjumping** (cf. *infra*).

14.6 Sophistiquer l'algorithme Backtrack

L'algorithme backtrack souffre de nombreux défauts, et des solutions variées ont été offertes et analysées dans la littérature (cf. (Gaschnig 1977; Haralick and Elliot 1980; Dechter and Meiri 1989; Dechter 1990; Prosser 1993; Schiex and Verfaillie 1993b; 1993a)...). Un premier point, usuel dans le traitement de problèmes NP-complets, a été la définition d'heuristiques à caractère général qui viennent normalement améliorer l'efficacité en moyenne de l'algorithme. Un second point consiste à modifier l'algorithme lui-même en tentant de détecter, à moindre coût et le plus rapidement possible, la non consistance globale de l'instanciation courante. On sait alors qu'il est impossible d'aboutir à une solution dans cette branche de l'arbre et on peut continuer l'exploration dans une autre branche. Les différentes stratégies existantes peuvent être séparées en :

1. **schémas rétrospectifs :** (ou *look-back schemes*) qui profitent de l'exploration arborescente effectuée durant l'exécution de l'algorithme, et en particulier des violations de contraintes, pour démontrer, à moindre coût, qu'une instanciation *strictement contenue* dans l'instanciation courante, et la plus petite possible, est non globalement consistante. Deux catégories d'algorithmes, finalement assez proches, rentrent dans ce cadre général :
   (a) Le *backtrack intelligent* utilise immédiatement la preuve qu'une instanciation $\mathscr{A}$, incluse dans l'instanciation courante, est non globalement consistante en effectuant un retour-arrière *non chronologique* qui supprime de l'espace

exploré, simplement (et sans consommation mémoire) une partie des instanciations qui contiennent $\mathscr{A}$.

(b) La *mémorisation de contraintes* (aussi appelée *apprentissage*) va plus loin en mémorisant dans le CSP le fait que $\mathscr{A}$ est non globalement consistante sous la forme d'une contrainte interdisant $\mathscr{A}$ (qui est *induite* par le CSP). Cette mémorisation s'ajoute au retour-arrière intelligent. Il faut alors faire un judicieux compromis entre la mémoire consommée et les effets bénéfiques de la mémorisation.

2. **les schémas prospectifs :** (ou *look-ahead schemes*) qui modifient le mécanisme d'exploration habituel de l'algorithme backtrack afin de détecter *a priori* les instanciations non globalement consistantes. On va donc déclencher une exploration « annexe » après chaque instanciation. Là encore, un judicieux compromis doit être effectué entre le coût d'une exploration plus large et les effets bénéfiques de celle-ci.

Comment comparer tous ces algorithmes expérimentalement ? Trois mesures *complémentaires* permettent de juger la qualité des algorithmes obtenus quant au traitement d'un CSP donné :

1. le nombre d'états développés permet de juger la taille de l'espace exploré. Cependant, et pour certains algorithmes du type *prospectif*, il y a plusieurs façons de définir ce qu'est un état ;
2. le nombre de vérifications de satisfaction de contraintes (*consistency checks*) permet de juger grossièrement les performances globales. Cette mesure a l'avantage d'être indépendante de l'implémentation, de la machine... Mais elle ne prend pas en compte le fait que, bien souvent, la mise en place d'une machinerie délicate, destinée à diminuer le nombre de vérifications de contraintes, peut entraîner un accroissement important des temps d'exécution ;
3. le temps d'exécution est la mesure la plus intéressante pratiquement, mais elle dépend de tous les paramètres cités précédemment ce qui lui retire un peu de sa valeur.

Une bonne évaluation doit donc idéalement comprendre la donnée des trois paramètres (en les définissant précisément) et sur un nombre important de problèmes différents. C'est une tâche lourde, qui peut être suivie d'une analyse statistique.

14.6.1 Ordres et heuristiques

Une analyse élémentaire du fonctionnement de l'algorithme backtrack montre que trois ordres influencent son exécution :

1. **un ordre d'instanciation des variables**, ou *ordre vertical*. Cet ordre peut avoir un impact très important sur l'efficacité de l'algorithme puisqu'il décide du moment à partir duquel la satisfaction d'une contrainte peut être vérifiée et influe donc sur la taille de l'espace exploré.

   Une heuristique est donc souvent introduite pour choisir un ordre vertical « pertinent ». On utilise habituellement une heuristique s'appuyant sur le principe de l'*échec d'abord* (*first fail principle*) qui cherche à faire apparaître le plus rapidement possible des violations de contraintes. Elle peut être utilisée *statiquement* (avant

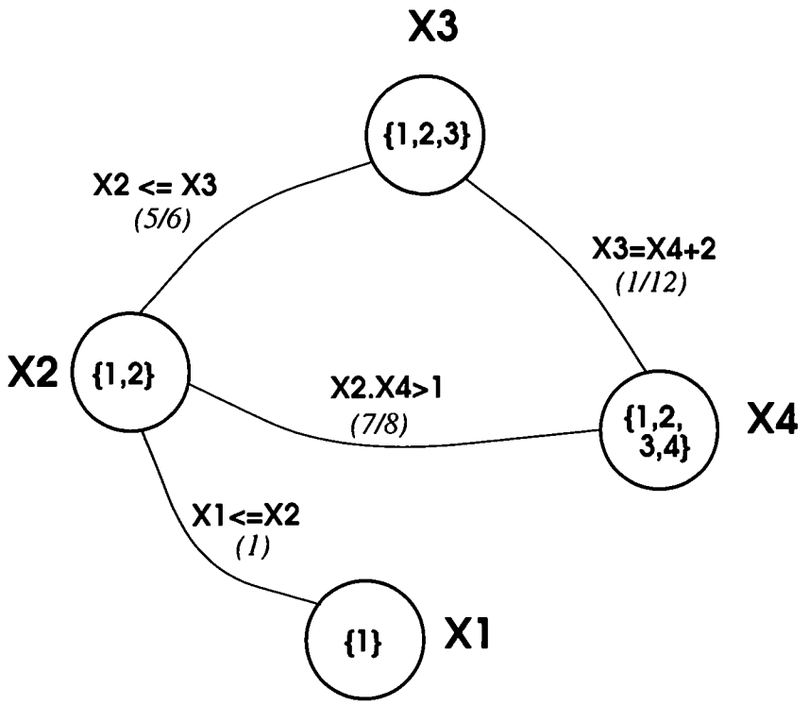


Figure 14.7 – Un CSP aux caractéristiques disparates

l'exécution) pour construire un ordre qui sera utilisé de la même façon dans toutes les branches ou *dynamiquement*, en calculant pendant l'exécution de l'algorithme, et à chaque nœud, la prochaine variable qui sera instanciée. Pour simplifier nos exemples et nos schémas d'algorithmes, nous supposerons qu'un ordre d'instanciation fixe des variables est utilisé.

Dans le cas statique, la meilleure des heuristiques ne prenant en compte qu'un seul critère semble être celle de *cardinalité maximum*. Elle consiste à choisir une première variable à instancier aléatoirement (en pratique, une variable de degré maximum est préférable). La prochaine variable liée sera celle reliée (par des contraintes) au plus grand nombre de variables déjà définies dans l'instanciation courante (Haralick and Shapiro 1979; 1980). D'autres éléments peuvent être considérés : variable de plus petit domaine, variable participant au plus grand nombre de contraintes, variable impliquée dans la contrainte la moins satisfiable, ordre de largeur minimale (cf. §14.7.1 pour la définition de la largeur d'un ordre).

Dans le cas dynamique, l'heuristique appelée *min-domain* ou *réarrangement dynamique* a longtemps été considérée comme un des meilleurs choix possibles. Elle consiste à choisir la variable ayant le nombre minimum de valeurs vérifiant une propriété de consistance locale donnée (Purdom 1983; Haralick and Elliot 1980; Nudel 1983; Dechter and Meiri 1989). La prise en compte simultanée d'autres caractéristiques (telles que le degré de la variable) permet en général une amélioration sensible des performances. (Bessière and Régin 1996) a ainsi popularisé l'utilisation du rapport entre ce nombre de valeurs et le degré de la variable.

Considérons le CSP dont le graphe est illustré à la figure 14.7, présenté dans (Mittal and Nadel 1990). Les contraintes sont définies par des prédicats et pour chacune on a indiqué, en italique et entre parenthèses, son taux de satisfiabilité. Ce CSP présente une grande disparité dans les tailles de domaine, les taux de satisfiabilité des contraintes et les degrés (nombre de contraintes pesant sur une même variable) et est donc idéal pour illustrer notre propos.

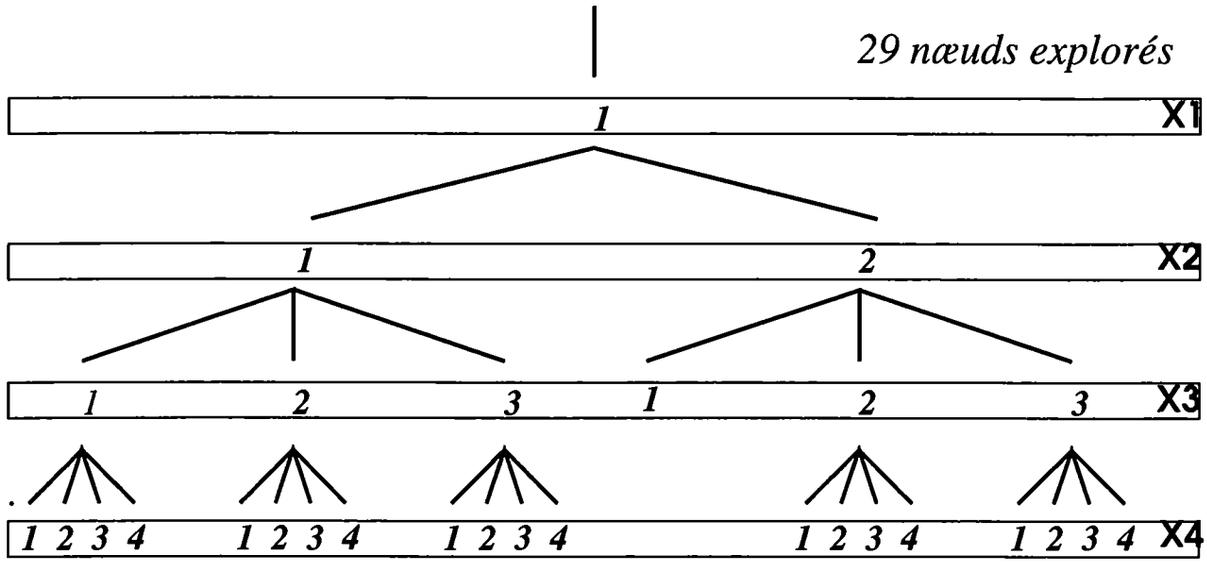


Figure 14.8 – Plus petits domaines d'abord

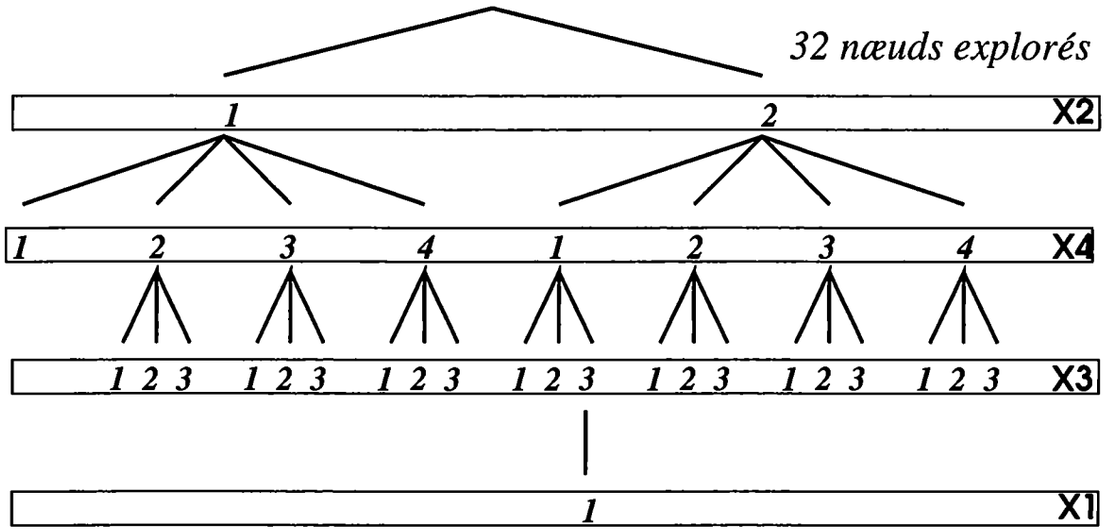


Figure 14.9 – Degré maximum d'abord

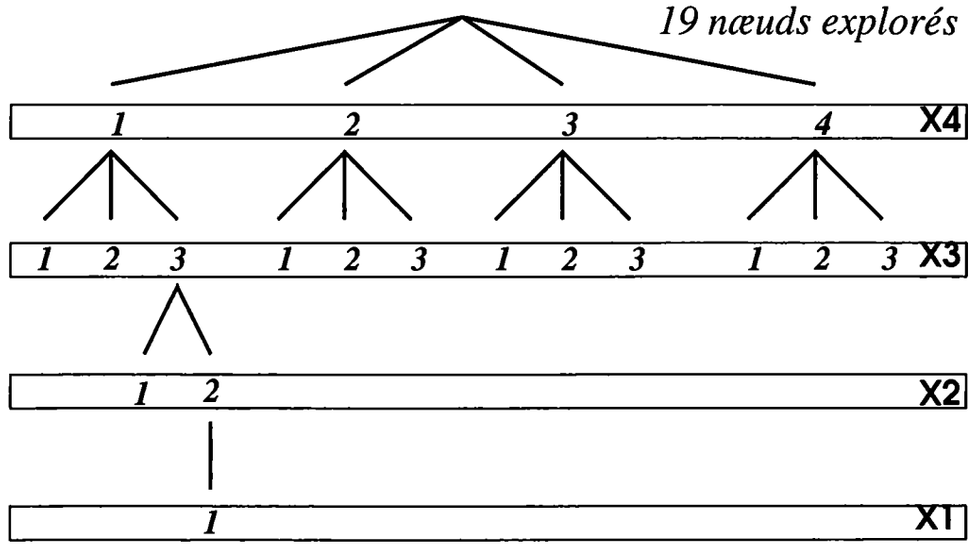


Figure 14.10 – Contrainte la moins satisfiable d'abord

Considérons trois heuristiques statiques d'ordre vertical :

(a) choix de la variable de plus petit domaine, définissant l'ordre d'instanciation (x_1, x_2, x_3, x_4) ;

(b) choix de la variable de degré maximum d'abord qui définit l'ordre d'instanciation (x_2, x_4, x_3, x_1) (par exemple) ;

(c) choix de la variable participant à la contrainte la moins satisfiable d'abord, définissant l'ordre (x_4, x_3, x_2, x_1) (par exemple).

Les arbres explorés lors de la recherche de toutes les solutions sont illustrés dans les figures 14.8, 14.9 et 14.10. Dans le cas présent, on observe que la meilleure heuristique est celle qui prend en compte la dureté des contraintes. En général, il est judicieux d'effectuer un compromis entre la taille des domaines, la structure du graphe de contraintes et les taux de satisfiabilité des contraintes.

L'introduction d'une heuristique « verticale » permet d'obtenir, en général, des gains très importants (sans, bien sûr, retirer au problème son caractère NP-complet).

2. **un ordre d'instanciation des valeurs** pour chaque variable ou ordre *horizontal*. Là encore, on peut choisir une heuristique statique ou dynamique. Il faut noter qu'une telle heuristique n'aura aucun effet sur l'efficacité de la recherche si le CSP est inconsistant ou si l'on cherche toutes les solutions puisqu'on devra alors explorer toute la largeur de l'arbre.

   Bien que quelques propositions d'heuristiques d'ordonnancement horizontal à caractère général aient été faites[23], il nous semble que les heuristiques d'ordre horizontal les plus efficaces sont dépendantes du domaine, les meilleures permettant de prendre en compte des caractéristiques globales des problèmes traités. En particulier, l'ordre d'instantiation des valeurs peut souvent être avantageusement guidé par le critère d'optimisation (s'il y en a un).

3. **un ordre de vérification des contraintes** à chaque nœud. Il n'apparaît pas explicitement dans la procédure backtrack, mais sera utilisé par la fonction qui vérifie si une instanciation est consistante. En effet, il est habituellement possible, après l'instanciation d'une nouvelle variable, de vérifier la satisfaction de plusieurs contraintes. L'ordre de vérification des contraintes n'influence pas la taille de l'espace exploré, mais permet de diminuer le nombre de test de satisfaction de contraintes (ou *consistency checks*). Le *first fail principle* conduit à vérifier en premier les contraintes les moins satisfiables.

Plus récemment, un certain nombre de travaux proposent d'utiliser directement les algorithmes de propagation pour évaluer quelle paire (variable, valeur) est la plus intéressante à affecter dans un contexte donné. L'idée générale est de choisir une valeur qui laisse le maximum de possibilités pour le futur et une variable qui réduit la taille de l'arbre de recherche le plus possible. Voir (Geelen 1992; Frost and Dechter 1995; Meseguer and Larrosa 1995). L'apport en terme d'efficacité de telles heuristiques n'est pas encore bien établi.

---

23 Heuristiques s'appuyant sur des modèles statistiques (Haralick and Shapiro 1980; Nudel 1983), heuristique de minimisation des conflits (Minton *et al.* 1990; 1992), surtout efficace, nous semble-t-il, pour les problèmes ayant une densité en solutions très importante.

14.6.2 Backtrack intelligent

L'idée directrice est d'exploiter les violations de contraintes constatées pendant l'exploration arborescente pour construire, plus ou moins localement, une instanciation *contenue* dans l'instanciation courante et non globalement consistante. On pourra alors reconsidérer les liaisons de plusieurs variables plutôt que de n'en reconsidérer qu'une (retour à l'état précédent) comme le fait l'algorithme backtrack.

Nous présenterons ici l'algorithme appelé **conflict based backjumping** dans (Prosser 1993), et qui est une spécialisation aux CSP du principe général de *retour-arrière dirigé par les dépendances* (cf. (Stallman and Sussman 1977; Doyle 1979; Bruynooghe and Pereira 1984; Schiex and Verfaillie 1993b; 1993a; Ginsberg 1993))

Le moment de choix pour tenter d'analyser les causes d'un échec afin d'éviter la redécouverte répétée de cet échec est le retour-arrière :

Définition 14.16 – *Lors d'une exploration arborescente, il y a retour arrière quand une instanciation consistante considérée à un nœud donné n'a pu être étendue en une solution.*

Il y a donc retour-arrière à chaque fois qu'une instanciation $\mathscr{A}$ non globalement consistante est considérée par backtrack. Le but du **conflict based backjumping** est de calculer, lors de chaque retour-arrière, un ensemble de variables, dites *variables de conflit* dont les valeurs dans $\mathscr{A}$ suffisent à expliquer le retour-arrière[24]. On sait qu'il est alors inutile de considérer toute instanciation partielle donnant à ces variables de conflit la même valeur que dans $\mathscr{A}$: il sera possible de reconsidérer immédiatement la valeur de la plus basse, dans l'ordre d'instanciation, de ces variables de conflit.

S'il y a retour-arrière, c'est parce que l'ensemble des instanciations *terminales* descendantes du nœud courant sont inconsistantes *i.e.*, violent une contrainte. De ce fait, toutes les variables instanciées dans $\mathscr{A}$ qui participent à une de ces contraintes violées sont les seules qui soient directement responsables du retour-arrière : elle forment un ensemble de variables de conflit.

Sur l'exemple d'application de l'algorithme **backtrack**, considérons l'instanciation partielle $\mathscr{A} = \{x_1 \rightarrow a, x_2 \rightarrow c, x_3 \rightarrow c, x_4 \rightarrow b, x_5 \rightarrow c\}$. Cette instanciation n'a pu être étendue en une instanciation complète et consistante du seul fait de la violation de la contrainte entre x_6 et x_3. x_3 est la seule variable instanciée dans $\mathscr{A}$ qui participe à cette contrainte : c'est une variable de conflit et il est donc inutile de considérer n'importe quelle instanciation contenant $\{x_3 \rightarrow c\}$ car elle ne pourra mener à une instanciation complète consistante : on peut donc effectuer un retour-arrière directement jusqu'au niveau de x_3 sans qu'une solution ne soit perdue[25].

L'algorithme correspondant, destiné à la recherche de la première solution seulement, est incarné dans la fonction récursive **CBJ**. Il fait appel à la fonction Consistante qui reçoit une instanciation et qui retourne l'ensemble des variables de la contrainte violée

---

24 Les algorithmes existants se distinguent surtout par la qualité de l'ensemble de variables de conflit construit. Certains (Dechter 1990) proposent même de calculer effectivement les ensembles minimaux au sens de l'inclusion. L'algorithme **conflict based backjumping** ne demande aucun test de contrainte supplémentaire par rapport au mécanisme de base de backtrack.

25 Il faut noter que les valeurs des variables intermédiaires x_4 et x_5 sont alors oubliées. Ce n'est pas le mécanisme qui est utilisé dans tous les algorithmes de retour-arrière intelligent. Dans certaines propositions récentes (Verfaillie 1993; Schiex and Verfaillie 1993b; Ginsberg 1993), ces variables conservent leurs valeurs et seule la valeur de la variable de conflit est reconsidérée.

si l'instanciation est inconsistante, $\varnothing$ sinon. La procédure s'applique à une séquence de variables V (initialement X) et à une instanciation $\mathscr{A}$ (initialement vide) :

Algorithme 6: Le Conflict Based BackJumping

```
CBJ(V, 𝒜);
si V = ∅ alors
    𝒜 est une solution;
    retourner ∅;
sinon
    soit x_i ∈ V, conflit = ∅, no-BJ = true;
    pour chaque ν ∈ d_i et tant que no-BJ faire
        soit conflit-local = Consistante(𝒜 ∪ {x_i → ν});
        si conflit-local = ∅ alors
            soit conflit-fils = CBJ(V − {x_i}, 𝒜 ∪ {x_i → ν});
            si x_i ∈ conflit-fils alors
                conflit ← conflit ∪ conflit-fils;
            sinon
                conflit ← conflit-fils;
                no-BJ = false;
        sinon
            conflit ← conflit ∪ conflit-local;
    retourner conflit;
```

L'application de l'algorithme est illustrée dans la figure 14.11. Au total 27 nœuds et 28 tests de consistance sont effectués. La petite taille du problème explique le faible gain obtenu[26].

De nombreux autres algorithmes du même type ont été proposés[27] : le *graph based backjumping*, ou retour-arrière basé sur la structure, présenté dans (Dechter 1990), le *backjumping*, explicité dans (Gaschnig 1979), le *dynamic backtracking* de (Ginsberg 1993), l'algorithme *local changes* présenté dans (Verfaillie 1993; Schiex and Verfaillie 1993b)...

14.6.3 Mémorisation de contraintes

L'idée directrice est d'engranger intelligemment de l'information qui servira durant la résolution, et qui pourra, de plus, servir ultérieurement lors de la résolution du même problème ou d'un problème plus contraint[28]. Les résultats principaux que

26 En comparaison avec l'algorithme **backtrack**, sans heuristique d'ordre vertical ou horizontal, nous avons pu constater des améliorations en termes de temps et de nombre de vérifications de satisfaction de contrainte, de plus de deux ordres de grandeur sur le simple problème du Zèbre (cf. 14.9).

27 Nous ne considérerons pas l'algorithme appelé *backmarking*, décrit dans (Gaschnig 1977). (Prosser 1993) considère qu'il s'agit plus d'un schéma prospectif que rétrospectif (et il peut effectivement s'hybrider avec des schémas rétrospectifs).

28 Des contraintes ont été ajoutées, des domaines ont été réduits.

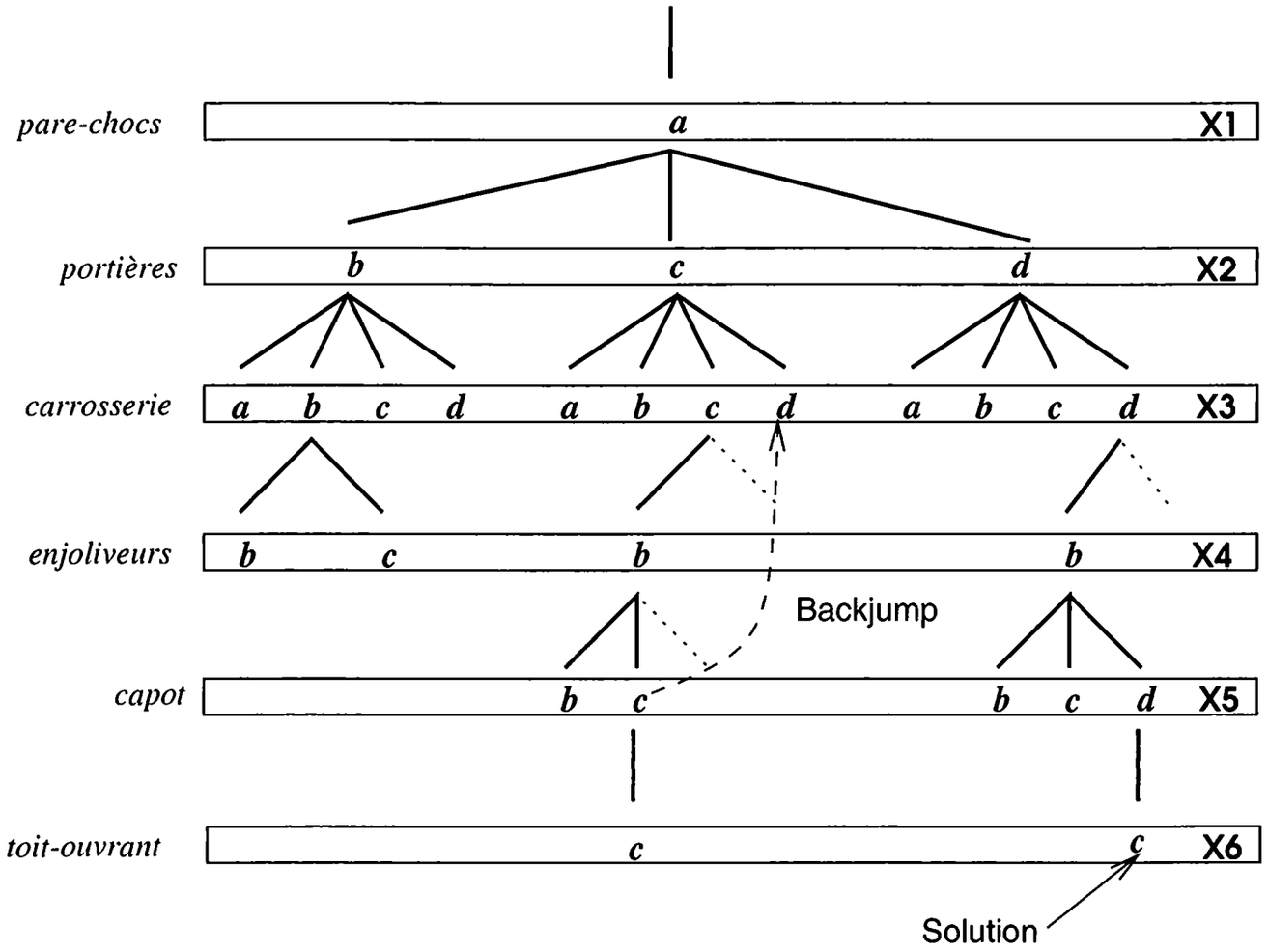


Figure 14.11 – Application de l'algorithme **CBJ**

nous allons exposer apparaissent dans (Dechter 1990; Schiex and Verfaillie 1993b; 1993a). La méthode est à la fois très proche du retour-arrière intelligent et du filtrage.

Reconsidérons l'instanciation $\mathscr{A} = \{x_1 \rightarrow a, x_2 \rightarrow c, x_3 \rightarrow c, x_4 \rightarrow b, x_5 \rightarrow c\}$ dans le cadre du retour-arrière intelligent : nous avions réussi à conclure que toute instanciation contenant l'instanciation $\{x_3 \rightarrow c\}$ ne pouvait mener à une solution (information appelée *nogood* par analogie avec les *nogoods* des ATMS). En sus du retour-arrière intelligent, qui consiste à tirer parti immédiatement de l'information, on peut incorporer cette information dans le CSP traité en retirant c du domaine de x_3[29]. On diminue ainsi potentiellement l'espace exploré. La figure 14.12 illustre l'arbre exploré si ce mécanisme est systématiquement appliqué. On est alors amené à explorer 24 nœuds et à vérifier la satisfaction de 25 contraintes. Les quatre instanciations non globalement consistantes détectées sont mises en évidence.

Ce mécanisme constitue un complément très utile du retour-arrière intelligent : bien souvent, le mécanisme utilisé dans l'algorithme CBJ permet de prouver qu'une instanciation partielle $\mathscr{A}$ est non globalement consistante, mais il ne peut en tirer parti, car une des variables de $\mathscr{A}$ n'est autre que la variable courante. On se limite alors à un retour-arrière classique. La mémorisation rend possible l'utilisation de cette information dans d'autres branches de l'arbre.

---

29 Comme le ferait l'établissement de l'arc-consistance. Dans le cas de la mémorisation de contraintes appliquée à l'algorithme CBJ on effectue un filtrage qui ne s'exprime pas naturellement en terme de k-consistance, même directionnelle.

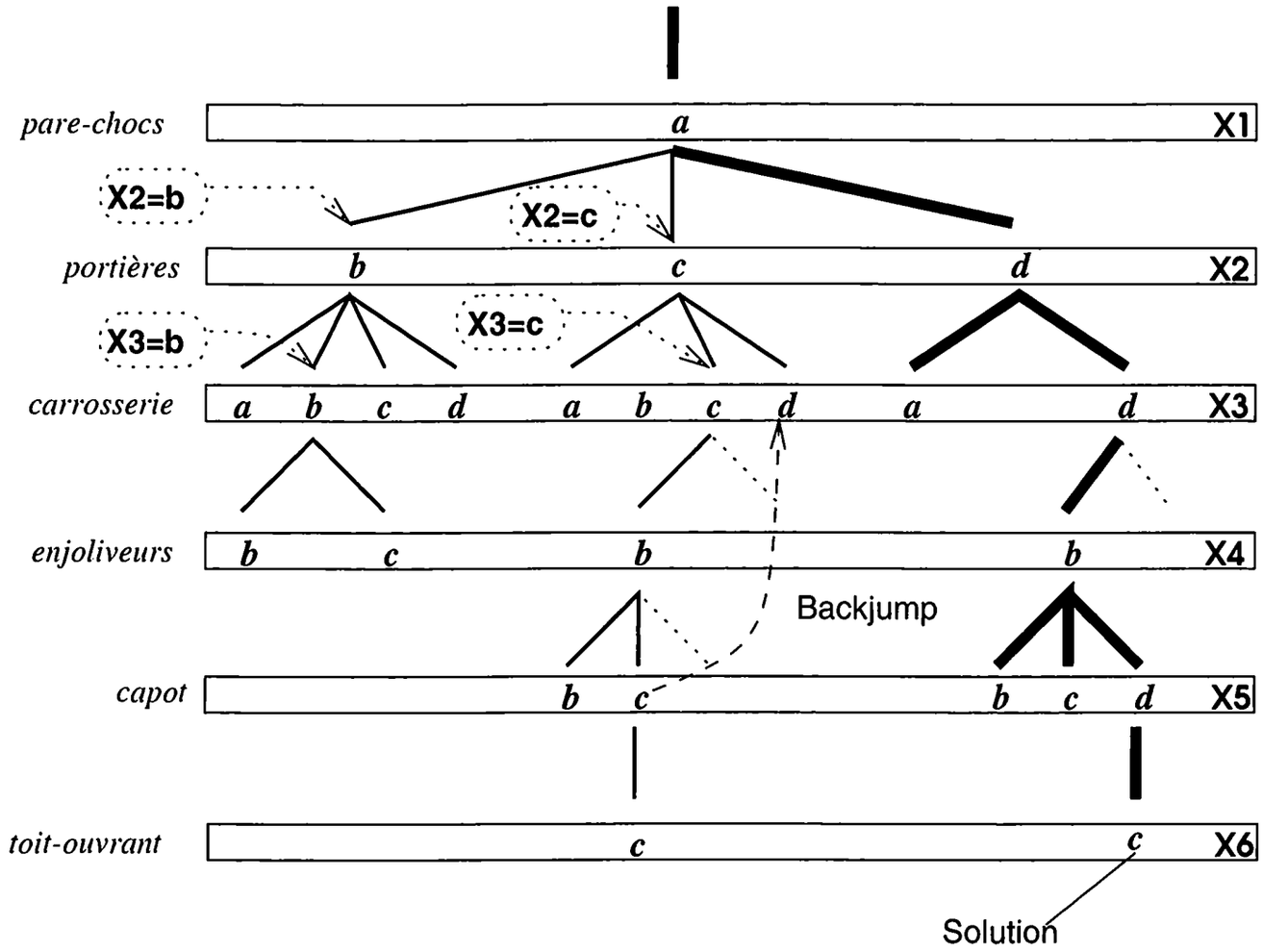


Figure 14.12 – Application de la mémorisation de contraintes

Un problème de taille est posé par le nombre potentiel d'instanciations non globalement consistantes que peut construire un algorithme de ce type : il croit exponentiellement avec le nombre de variables. En général, le choix effectué consiste à ne mémoriser que les instanciations non globalement consistantes de cardinal inférieur à k. On mémorise ainsi moins d'instanciations, mais aussi les plus potentiellement intéressantes : une instanciation non globalement consistante de cardinal limité a plus de chances d'être incluse dans d'autres instanciations. On parle ainsi d'apprentissage du premier ordre (mémorisation des instanciations de cardinal un), du deuxième ordre, etc. Il faut noter l'avantage très net de l'apprentissage du premier ordre, qui ne modifie pas les contraintes, mais les domaines, et qui est, de ce fait, d'une mise en œuvre simplifiée. Les expérimentations semblent tout de même montrer que la mémorisation du deuxième ordre permet d'obtenir un gain notable (Dechter 1990; Schiex and Verfaillie 1993b; 1993a).

Tout comme le filtrage, il faut noter les bonnes propriétés des algorithmes de mémorisation de contraintes : les contraintes inférées étant induites par le CSP, il est possible de les mémoriser ou non à volonté. Enfin, si un CSP (X, D, C) a été résolu, toutes les contraintes induites inférées restent utilisables directement pour résoudre un CSP (X, D, C') avec $C \subset C'$. Dans le cas de notre exemple, une nouvelle résolution du même CSP entraîne l'exploration de 9 nœuds et 9 contraintes seulement, sans aucun retour-arrière (sous-arbre en gras dans la figure 14.12). Dans le cas des retraits de contraintes, un mécanisme plus fin doit être mis en place (cf. §14.8 et (Schiex and Verfaillie 1993b; 1993a)).

14.6.4 Méthodes prospectives

Ce sont, en général, les méthodes les plus efficaces. La majorité des langages et systèmes logiciels s'appuyant sur le cadre CSP (en particulier les langages de programmation par contraintes) utilisent ces techniques pour résoudre le problème de satisfaction.

L'algorithme **backtrack** est simplement modifié en remplaçant la vérification de la consistance de l'instanciation partielle courante par l'établissement d'une propriété de consistance locale. Il y a retour-arrière dès que la propriété de consistance locale n'est plus vérifiée. Il est alors nécessaire de « défaire » les effets du filtrage effectué précédemment.

L'essentiel des algorithmes utilisés s'appuient sur les versions plus ou moins « dégradées » de la consistance d'arc que nous avons présentées dans la section 14.5.2. Chaque instanciation peut donc supprimer des valeurs des domaines d'autres variables. Il y a retour-arrière dès qu'un domaine est vide. Selon le niveau d'arc-consistance utilisé, on aboutira aux algorithmes :

- *backtrack* ou *test and generate* : emploi d'un *check backward* après chaque liaison ;
- *forward-checking* : emploi d'un *check forward* après chaque liaison ;
- *partial look-ahead* : emploi d'un *partial look future* après chaque liaison ;
- *full look-ahead* : emploi d'un *look future* après chaque liaison ;
- *real full look-ahead* : établissement de la consistance d'arc complète après chaque instanciation[30].

Dans les quatre derniers cas, on parle alors d'algorithmes de type *choix et propagation*, qui permettent d'obtenir une efficacité très supérieure à celle du classique **backtrack**. Le choix du niveau de consistance locale utilisé n'est pas innocent et a une influence importante sur l'efficacité : un niveau de consistance faible entraîne l'exploration d'un arbre de grande taille mais le travail effectué à chaque nœud est faible, un niveau plus fort peut réduire la taille de l'arbre de recherche mais le travail effectué à chaque nœud est plus important.

À la suite de tests très limités et exposés dans (Haralick and Elliot 1980), le *forward-checking* a longtemps été considéré comme le compromis le plus intéressant. Il est maintenant largement admis que l'établissement de la cohérence d'arc complète à chaque nœud est un meilleur choix (que la majorité des langages de programmation avec contraintes avaient d'ailleurs fait depuis longtemps déjà). L'algorithme *real full look-ahead* a été sophistiqué pour exploiter finement les versions récentes du filtrage par arc-consistance (AC4, AC6, AC7 et AC-Inférence) pour aboutir aux algorithmes de Maintien de l'Arc-Consistance MAC4, MAC6, MAC7 et MAC-Inférence (Sabin and Freuder 1994; Bessière *et al.* 1995). La phase d'initialisation des algorithmes ACn est effectuée une fois seulement (à la racine de l'arbre de recherche). L'implémentation efficace de ces algorithmes est une tâche délicate (cf. (Régin 1995)).

L'algorithme Forward-checking est nettement plus simple à réaliser. Il est incarné dans la procédure récursive Forward-checking qui prend en argument une séquence de variables V à instancier (initialement X en entier) et une instanciation $\mathscr{A}$ (initialement vide). Elle utilise une procédure auxiliaire appelée Check-Forward. Nous supposerons que la procédure Sauvegarde, qui prend en argument une séquence de variables, empile

---

30 Il règne une certaine confusion sur le niveau précis de consistance locale utilisé par l'algorithme *full look-ahead* : *look-future* ou arc-consistance totale ? Nous reprenons ici la distinction proposée par (Nadel 1988).

les domaines des variables de la séquence sur une pile et que la procédure Restauration les dépile.

Algorithme 7: Le Forward-cheking

```
Forward-checking(V, 𝒜);
si V = ∅ alors
    𝒜 est une solution;
sinon
    soit x_i ∈ V;
    pour chaque ν ∈ d_i faire
        Sauvegarde(V − {x_i});
        si Check-Forward(x_i, ν, V) alors
            Forward-checking(V − {x_i}, 𝒜 ∪ {x_i → ν});
        Restauration(V − {x_i});
```

Algorithme 8: Check-Forward

```
Check-Forward(x_i, ν, V);
soit consistant = true;
pour chaque x_j ∈ V − {x_i} tant que consistant faire
    pour chaque ν' ∈ d_j faire
        si {x_i → ν, x_j → ν'} est non consistante alors
            d_j ← d_j − {ν'};
    si d_j = ∅ alors consistant ← false;
retourner consistant;
```

L'application de l'algorithme au même exemple que précédemment est illustrée figure 14.13. L'arbre exploré ne comporte plus que 10 nœuds, mais il aura tout de même fallu 31 tests de satisfaction de contraintes : le problème est de trop petite taille pour que la méthode ait un intérêt.

Il faut noter que l'utilisation de ces algorithmes renforce l'intérêt d'une heuristique d'ordre vertical du type *réarrangement dynamique* puisque la propriété de consistance établie à chaque nœud offre gratuitement des informations supplémentaires (domaines réduits).

14.6.5 Remise en cause de l'ordre d'exploration

Une fois les heuristiques d'ordonnancement vertical et horizontal choisies, l'ordre dans lequel les algorithmes qui s'appuient sur une mécanique de type backtrack vont explorer l'arbre des instanciations est, aux éventuels ex-æquo près, fixé. Si l'heuristique horizontale employée est réellement informée, cet ordre présente plusieurs faiblesses :

- il ne respecte absolument pas un éventuel ordre induit par l'heuristique horizontale sur les instanciations. La mécanique de l'algorithme impose en effet d'épuiser les valeurs d'une variable avant d'effectuer un retour-arrière. Parmi ces valeurs se

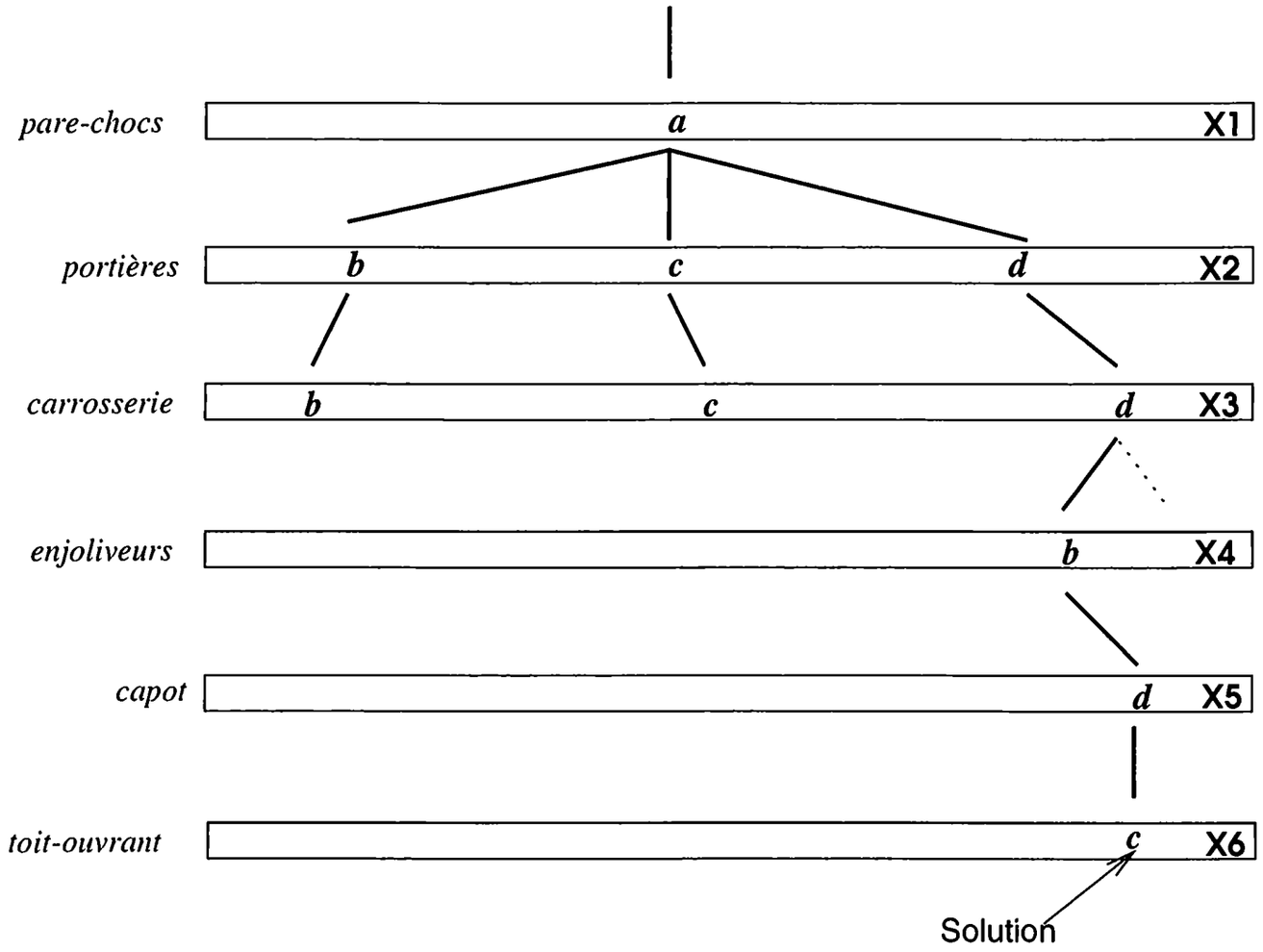


Figure 14.13 – Application de l'algorithme **forward-checking**

trouvent probablement des valeurs considérées comme médiocres par l'heuristique d'ordre horizontale et l'on aimerait pouvoir considérer les instanciations utilisant ces valeurs plus tardivement dans la recherche. Inversement, on aimerait pouvoir plus rapidement considérer les valeurs prometteuses de niveaux supérieurs ;

– le parcours en profondeur d'abord sera très long à reconsidérer les premiers choix effectués dans l'arbre. Or, ces premiers choix sont souvent ceux pour lesquels l'heuristique est la plus mal informée.

Ces restrictions sont absentes des techniques dites de recherche locale (recuit simulé, recherche tabou, GSAT..., cf. (Aarts and Lenstra 1997)). Différentes approches permettent, à un certain degré, de s'en affranchir sans perdre la complétude. Elles présentent surtout un intérêt pour la résolution de problèmes consistants et assez sous-contraints pour lesquels une exploration d'une fraction de l'arbre de recherche devrait suffire à trouver une solution. La plus connue est probablement la *Limited Discrepancy Search* (LDS) introduite par (Harvey and Ginsberg 1995). Étant donné une heuristique horizontale, on appelle *désaccord* (*discrepancy*) tout nœud de l'arbre de recherche pour lequel la valeur choisie pour la variable courante n'est pas celle conseillée par l'heuristique horizontale. L'algorithme LDS est un algorithme itératif qui, à l'itération k va considérer les instanciations ayant $k-1$ désaccords. Pour illustrer son fonctionnement, considérons un CSP avec des domaines de taille 2, 4 variables et supposons que l'heuristique horizontale conseille systématiquement de prendre la première valeur du domaine (à gauche). Les branches explorées à chaque itération sont visibles dans la figure 14.14.

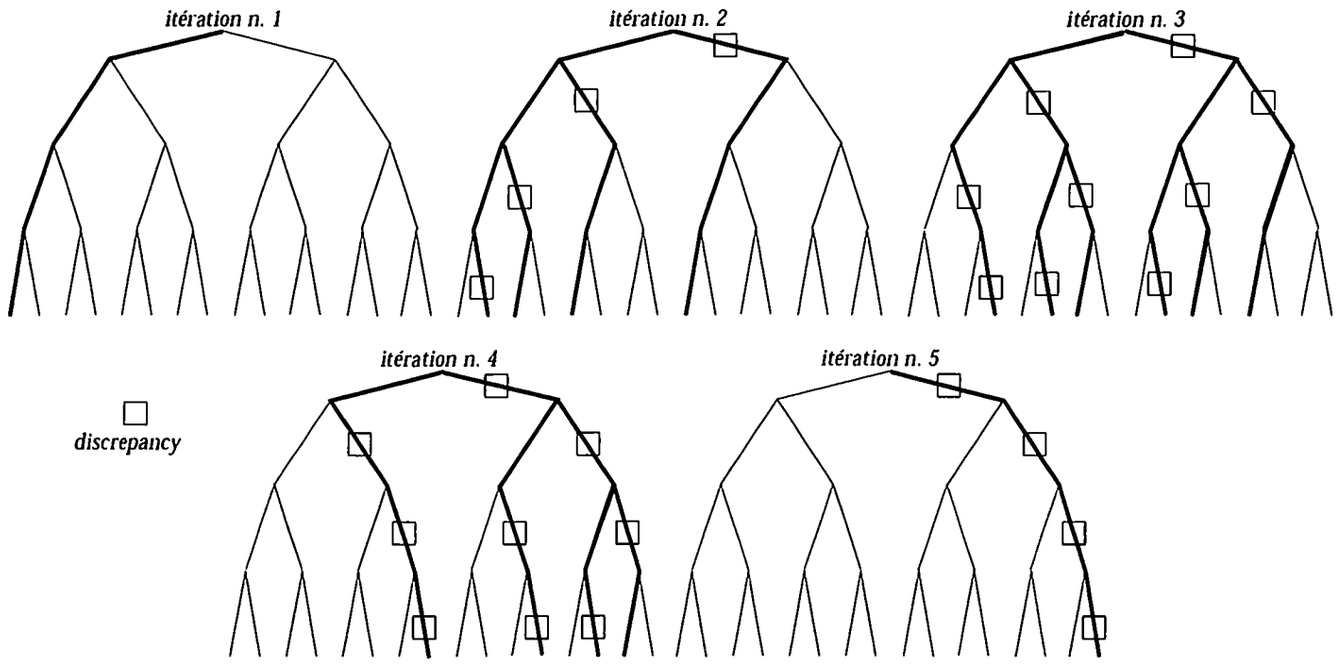


Figure 14.14 – Parcours en « désaccords limités », les branches explorées à chaque itération sont visibles en gras.

Cette méthode a été sophistiquée par (Korf 1996; Walsh 1997). Utilisée comme support d'une méthode prospective comme le forward-checking ou MAC, elle induit un surcoût important en terme de temps de filtrage par nœuds développé car l'effort de filtrage entre deux branches explorées successivement n'est pas, comme pour backtrack, partagé du fait d'un préfixe identique.

Une façon simple de remedier à ce coût tout en garantissant une bonne diversification de la recherche,introduite sous le nom de *interleaved depth first search* consiste à simuler, sur un seul processeur, la recherche en parallèle sur k processeurs, dans k sous-arbres de l'arbre de recherche (Meseguer 1997; Meseguer and Walsh 1998).

14.6.6 Méthodes hybrides

Il est souvent possible d'hybrider les algorithmes que nous avons évoqués. Les gains obtenus ne sont, hélas, pas égaux au produit des gains des méthodes hybridées, mais peuvent être intéressants. Un nombre important de telles hybridations sont définies et évaluées dans (Prosser 1993). Une hybridation mêlant retour-arrière intelligent, mémorisation de contraintes et *forward-checking* est également proposée et évaluée dans (Schiex and Verfaillie 1993b; 1993a).

Comment choisir parmi tous ces algorithmes ? Le fait est que, pour le traitement de problèmes quelconques *a priori*, les algorithmes prospectifs de type MAC fournissent les meilleurs résultats. L'introduction de sophistications supplémentaires, quand elle est bénéfique, permet des gains d'autant plus importants que le problème est de grande taille.

Le point important reste qu'il est extrêmement difficile d'estimer *a priori* l'efficacité de n'importe lequel de ces algorithmes face à un problème donné. Cependant, une méthode statistique d'échantillonnage, introduite dans (Knuth 1975) et affinée dans (Chen 1992; Lobjois and Lemaître 1998) permet de définir un estimateur non biaisé de la taille de l'arbre développé par un algorithme de type backtrack sur un problème donné. La variance de cet estimateur peut, malheureusement, être très importante.

14.7 Classes polynomiales et décomposition

Nous avons décrit, dans la section 14.5, un ensemble de propriétés de consistance locale, dont l'établissement, via un algorithme de filtrage, s'effectue en temps polynomial. Pour certaines classes de CSP, une fois un certain filtrage effectué, le problème de satisfaction devient lui aussi polynomial. Le problème de satisfaction du CSP initial est alors polynomial. Nous nous proposons de rapidement présenter les principales classes de ce type exhibées à ce jour.

On peut les séparer rapidement en deux ensembles : les classes caractérisées uniquement par la structure du graphe (ou hyper-graphe) du CSP et les classes caractérisées par le contenu précis des relations (ou des domaines) du CSP, ou *micro-structure* du CSP.

14.7.1 Classes polynomiales structurelles

Directement ou indirectement, la majorité de ces classes polynomiales découlent de propriétés générales classiques sur les graphes ou hypergraphes liant structure et applicabilité d'algorithmes du type *programmation dynamique* (Bertelé and Brioshi 1972). Une première classe polynomiale a été identifiée et caractérisée via la notion de largeur (cf. (Freuder 1982; 1985; 1988)) du graphe d'un CSP binaire :

Définition 14.17 – Largeur – *Étant donnés un CSP binaire et un ordre sur ses variables, la largeur d'une variable x_i est égale au nombre de variables connectées à x_i et qui la précèdent dans l'ordre. La largeur d'un ordre est le maximum de la largeur des variables dans cet ordre. La largeur d'un CSP est égale à la largeur minimale sur tous les ordres.*

La largeur d'un CSP se calcule simplement (via un algorithme linéaire dans la taille du graphe du CSP (Freuder 1982)). Elle correspond, très informellement, à la connexité du graphe : un arbre est de largeur 1, un graphe complet de n nœuds a une largeur de $(n-1)$.

Théorème 14.1 – (Freuder 1982) – *Étant donné un CSP de largeur W, si le niveau de forte consistance du CSP est supérieur à W, alors il existe un ordre d'instanciation (celui de largeur minimale) qui est glouton* (i.e., *qui permet de résoudre le CSP sans retour-arrière).*

Ce théorème n'a pas un impact aussi important qu'il pourrait paraître au premier abord. En effet, ce n'est pas parce que l'on a un CSP de largeur $(k-1)$ que le problème de satisfaction pourra être résolu en $O(n^k.d^k)$ via l'établissement de la forte k-consistance : dès lors que k est plus grand que 2, le filtrage peut ajouter des contraintes au CSP et peut donc modifier sa largeur. Les seuls CSP pour lesquels le théorème a un intérêt immédiat sont les CSP de largeur 1 (dont les graphes sont des arbres), car il suffira de les rendre arc-consistants, ce qui modifie les domaines, mais n'ajoute pas de contraintes. Il s'agit en fait du résultat essentiel.

Corollaire 14.1 – *Tout CSP binaire dont le graphe est arborescent peut être résolu en $O(n.d^2)$.*

En effet, l'algorithme **backtrack**, quand il utilise un ordre glouton, aboutit à une solution en un temps en $O(n.d)$, négligeable devant les $O(n.d^2)$ nécessaires à l'établissement de l'arc-consistance (le CSP étant arborescent, il a $m = (n-1)$ contraintes).

D'autres classes polynomiales structurelles ont été identifiées par la suite. Chacune de ces classes est caractérisée par un paramètre p sur le graphe ou hyper-graphe du CSP

et un algorithme dedié qui permet de résoudre le CSP en un temps polynomial dans le paramètre p. On peut normalement utiliser ces résultat en deux étapes : on calcule la valeur du paramètre p pour le graphe du CSP à résoudre puis on applique l'algorithme dédié. La majorité de ces paramètres définisse malheureusement un problème de calcul NP-difficile et l'on se contente dans ce cas d'un majorant fourni par une heuristique.

Dans un ordre chronologique, les paramètres caractéristiques identifiés sont la longueur de front minimum (*minimum front length* (Seidel 1981)), la taille de la plus grande composante biconnexe (Freuder 1985), la hauteur de pseudo-arbre (*pseudo-tree height* (Freuder and Quinn 1985)), le nombre du coupe-cycle (*cycle cutset number* (Dechter and Pearl 1987; Dechter 1990)), la largeur induite (*induced width* ou *tree-width*) (Dechter and Pearl 1989)), la largeur de bande (*bandwidth* (Zabih 1990)), le nombre du k-arbre (*k-tree number* (Freuder 1990)) et le degré de cyclicité (*degree of cyclicity* (Gyssens *et al.* 1994)). Seuls le calcul de la taille de la plus grande composante biconnexe et le degré de cyclicité sont des problèmes polynomiaux. Tous ces paramètres sont liés entre eux et on montre en particulier que la largeur induite et le nombre du k-arbre sont égaux. Le lecteur est invité à se reporter aux références pour une définition précise de chacun d'entre eux.

Les algorithmes qui permettent de résoudre le CSP une fois la valeur d'un paramètre p calculée (ou majorée) sont de deux grands types : les algorithmes qui effectuent suffisamment d'instanciations pour se ramener à un ensemble de problèmes polynomiaux et les algorithmes qui s'appuient sur la programmation dynamique : on résout complètement un ensemble de sous-problèmes et l'on fusionne les résultats pour obtenir une solution globale. Examinons un représentant de chacune de ces classes.

Coupe-cycle

L'algorithme du coupe-cycle est présenté dans (Dechter and Pearl 1987). L'idée est la suivante : si l'on a un CSP binaire qui n'est pas acyclique, on va couper chacun de ses cycles en instanciant une variable quelconque de chaque cycle. On obtient ainsi un CSP arborescent que l'on résoudra polynomialement. Si l'on trouve une solution, on a fini. Sinon, il faut revenir sur la valeur des variables instanciées. Plus précisément :

- on calcule un ensemble de variables $Y \subset X$ qui une fois instanciées supprimeront les cycles du CSP. Cet ensemble, dit *ensemble coupe-cycle*, devrait être le plus petit possible puisqu'il sera peut-être nécessaire de revenir sur les valeurs des variables de cet ensemble. La taille d'un ensemble coupe-cycle de cardinal minimum définit le paramètre de la méthode. Malheureusement, son calcul est un problème NP-difficile. On se contente donc de l'emploi d'une méthode heuristique sous-optimale ;
- on instancie les variables de Y avec un premier n-uplet de valeurs de leur domaine ;
- on résout le CSP arborescent ainsi obtenu par établissement d'une consistance locale (ici, l'arc-consistance, mais une version directionnelle, orientée à partir des feuilles de l'arbre suffit) ;
- si le CSP est inconsistant, on reconsidère les valeurs des variables de l'ensemble coupe-cycle.

La méthode reste exponentielle dans la taille de l'ensemble coupe-cycle. Elle permet tout de même de fournir une borne intéressante à la complexité de résolution du problème. Pour un CSP binaire de n variables dont les domaines sont de taille n et pour un ensemble coupe-cycle de cardinal c, il faudra établir l'arc consistance d^c fois dans le pire des cas. On

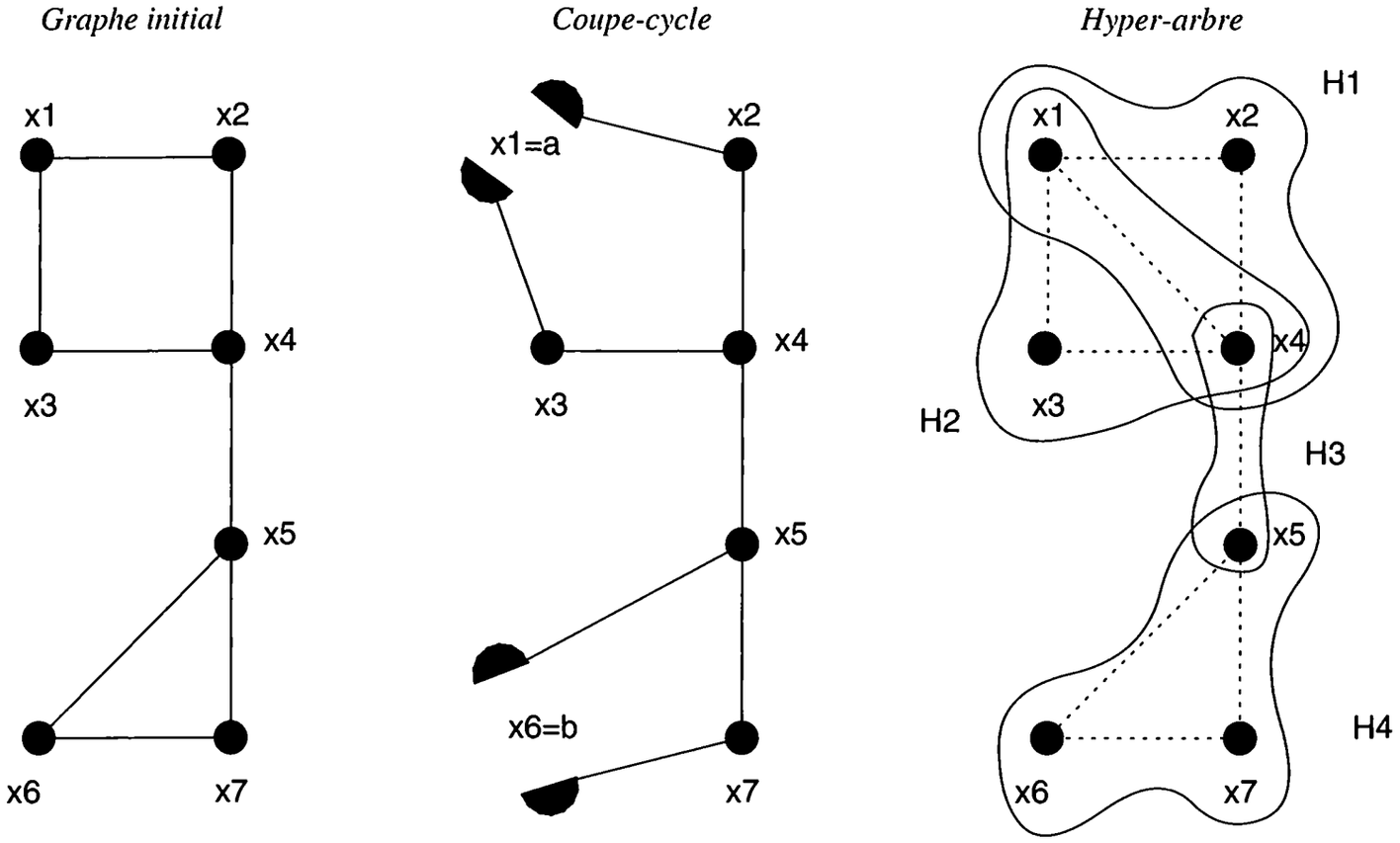


Figure 14.15 – Coupe-cycle et recouvrement par hyper-arbre.

aboutit donc à une complexité en $O(nd^{c+2})$, l'arc-consistance s'établissant en $O(n.d^2)$ sur un CSP arborescent.

Regroupement en hyper-arbre

La deuxième méthode est plus complexe et permet de traiter des CSP non binaires (nous nous limiterons ici à des CSP binaires pour des raisons de simplicité). On va recouvrir des groupes de contraintes formant des cycles par des hyper-arêtes de façon à former un CSP dont l'hyper-graphe est acyclique (Berge 1970; 1987) et qui pourra être résolu en un temps polynomial. De façon très grossière, la méthode se déroule de la façon suivante :

- triangulation du graphe de contraintes i.e., ajout d'arêtes au graphe de contraintes jusqu'à ce qu'il soit triangulé (ou chordal[31]). Un algorithme efficace (Tarjan and Yannakakis 1984) existe pour résoudre ce problème. Mais la triangulation obtenue ne comprend pas nécessairement le nombre minimum d'arêtes. On obtient ainsi une « bonne » complétion chordale du graphe d'origine.
- calcul des cliques maximales[32] du graphe triangulé. Chaque clique définit une hyper-arête correspondant à une contrainte n-aire portant sur les variables recouvertes et dont la relation associée est définie par l'ensemble des solutions du sous-CSP défini par les variables de l'hyper-arête ;

31 Tout cycle du graphe de longueur supérieure ou égale à 4 a une arête qui joint deux sommets non-consécutifs

32 Une clique est un sous-ensemble des sommets du graphe définissant un sous-graphe complet. La maximalité est au sens de l'inclusion. Un algorithme efficace permettant de résoudre le problème existe dans ce cas précis (Gavril 1972).

- résolution, par une méthode quelconque, de chacun des sous-CSP définis par les hyper-arêtes ;
- résolution du CSP n-aire acyclique par renforcement de consistance locale (cf. (Dechter and Pearl 1988c)).

La méthode de résolution reste exponentielle dans la taille de la plus grande hyper-arête générée (la plus grande clique). La triangulation puis la recherche de clique maximale dans le graphe triangulé fournissent en fait un majorant calculable efficacement du paramètre qui caractérise la méthode (la largeur induite). La méthode permet de fournir une nouvelle borne intéressante à la complexité de résolution du problème. Bien que cela ne soit pas évident au premier abord, l'établissement de la consistance adaptative (Dechter and Pearl 1988a) est en fait une version améliorée du *tree-clustering*. Une méthode hybride, utilisant coupe-cycle et recouvrement en arbre (sans triangulation) a été proposée dans (Jégou 1990) sous le nom de *cyclic clustering*. Elle permet d'exploiter plus finement la structure du graphe du CSP initial.

14.7.2 Classes polynomiales micro-structurelles

Le premier résultat remonte à (Montanari 1974). Il établit que la chemin consistance suffit pour garantir qu'un réseau est globalement consistant et minimal (décomposable) si les relations sont toutes monotones. Ce résultat a été affiné par la suite (van Beek 1992a) en une classe qui comprend en particulier les CSP dont les domaines ont un cardinal de 2, les CSP formés de contraintes fonctionnelles ou monotones (ce résultat généralise le résultat simlaire de (van Hentenryck *et al.* 1992) et la consistance d'arc), mais aussi les contraintes qui résultent d'une conjonction d'inégalités linéaires (strictes ou non) d'arité 2. Ce résultat est intéressant pour le raisonnement temporel (voir aussi (Dechter *et al.* 1991; van Beek 1989; 1992b; Deville *et al.* 1997)).

Un résultat général, présenté dans (Dechter 1992), et récemment affiné dans (van Beek and Dechter 1997) s'applique dans le cadre de CSP quelconques :

Théorème 14.2 – *Un CSP dont les domaines sont tous de cardinal inférieur à d, dont les contraintes sont toutes d'arité inférieure à a et qui est $(d.(a-1)+1)$ fortement consistant est aussi globalement consistant.*

D'autres classes ont été identifiées :

- la classe polynomiale des CSP binaires ayant « suffisamment » de contraintes fonctionnelles (cf. (David 1992; 1993)) ;
- la classe polynomiale des CSP dont les contraintes sont toutes « *Zero/One/All* » (cf. (Cooper 1994))...

14.8 Extensions du cadre classique

Le cadre CSP impose de modéliser tout problème sous la forme d'un réseau *figé* de contraintes binaires *dures*. Dans la pratique, de nombreux problèmes réels nécessitent, entre autres, l'expression de préférences ou la gestion d'un réseau de contraintes évoluant dans le temps contenant des contraintes d'arité quelconque. Une partie importante de l'activité de recherche en CSP s'effectuent maintenant dans ces directions. Nous les présentons rapidement.

14.8.1 Contraintes globales

La notion de contrainte « globale » est habituellement attachée à la définition d'un algorithme de filtrage dédié permettant de traiter efficacement un ensemble de contraintes spécifiques impliquant un nombre de variables quelconque. En effet, la majorité des travaux algorithmiques ou théoriques sur les CSP se limitent souvent au traitement de contraintes binaires. Cette limitation se fait sans perte de généralité en théorie car l'on sait efficacement transformer tout CSP contenant des contraintes d'arité supérieure à 2 en un CSP binaire (du moins tant que les contraintes sont exprimées en extension ce qui est rare pour les contraintes d'arité importante). En pratique, du fait de leur caractère local, les propriétés de cohérence locale (arc-consistance par exemple) ont un comportement différent selon que l'on travaille sur une représentation binaire ou sur la représentation originale, non binaire.

Considérons par exemple un CSP $\mathcal{P}_1$ défini par k variables, dont les domaines sont tous égaux à $\{1, \ldots, k-1\}$ et sur lesquelles pèse une unique contrainte de différence généralisée qui implique les k variables et qui stipule que ces variables sont toutes différentes[33]. Ce problème est bien sûr inconsistant. Une représentation binaire équivalente de ce CSP peut-être obtenue en remplaçant la contrainte de différence généralisée par $\frac{k.(k-1)}{2}$ contraintes binaires de différences, une contrainte existant entre chaque paire de variable. On obtient ainsi le CSP $\mathcal{P}_2$ équivalent et donc lui aussi inconsistant. On suppose que les contraintes sont définies par des prédicats.

Considérons la propriété d'arc-consistance dans les deux cas. Dans le problème $\mathcal{P}_2$, toutes les valeurs sont viables, le problème est arc-consistant. *A contrario*, dans $\mathcal{P}_1$, aucune valeur n'est viable : il est impossible de trouver dans la relation associée à la contrainte de différence généralisée un n-uplet qui implique une valeur puisque, dans ce cas extrême, elle n'en contient aucun. L'arc-consistance permet donc de détecter l'incohérence dans $\mathcal{P}_1$, elle n'a aucun intérêt pour traiter $\mathcal{P}_2$.

On voit donc, sur ce cas extrême, pathologique, l'intérêt que l'on peut avoir à traiter une contrainte non binaire sous sa forme initiale. Malheureusement, la généralisation des algorithmes d'établissement de l'arc-consistance à des contraintes d'arité quelconques aboutit à des algorithmes dont la complexité croît exponentiellement avec l'arité des contraintes (Mohr and Masini 1988; Bessière and Régin 1997). Si cette complexité reste théoriquement bonne quand les contraintes sont données en extensions (car la taille du problème croît elle aussi exponentiellement avec l'arité des contraintes), ce n'est plus le cas lorsque les contraintes sont définies par des prédicats, cas le plus fréquent pour les contraintes d'arité importante.

Il devient intéressant alors d'isoler des classes de contraintes d'arité quelconque, pratiquement intéressantes et pour lesquels il est possible d'établir une consistance locale généralisée à coût limité. C'est ce qui a été fait pour la contrainte de différence généralisée dans (Régin 1994). D'autres types de contraintes « globales » ont également profité d'algorithmes dédiés : contrainte de cardinalité (Régin 1996), *edge-finding* (Nuijten 1994) pour le traitement de contraintes issues du problème d'ordonnancement, contrainte de tri (Guernalec and Colmerauer 1997)... Les langages de programmation par contraintes (Chip, Ilog Solver...) sont habituellement généreusement équippés de tels algorithmes.

---

33 Ce CSP est bien sûr un représentant du fameux *problème des pigeons* : placer n pigeons dans $n-1$ boîtes dans que deux pigeons se retrouvent dans la même boîte.

14.8.2 Satisfaction partielle

La grande majorité des problèmes réels modélisables sous forme de CSP font généralement apparaître une partition des « contraintes » en deux catégories :

- des contraintes généralement liées à des aspects physiques (temporel, géométrique, énergétique...) qui ne doivent en aucun cas être violées ;
- des contraintes le plus souvent liées aux agents humains et exprimant des préférences personnelles. Elles doivent être *préférablement* satisfaites.

Or le cadre CSP ne permet d'exprimer que des contraintes nécessairement satisfaites. Cette limitation peut aboutir à deux états :

- si l'utilisateur choisit d'exprimer ses préférences sous forme de contraintes, il risque de produire un réseau de contraintes sans solution, alors que l'expression de préférences aurait permis l'élaboration d'une ou plusieurs solutions « les moins mauvaises possible » ;
- si l'utilisateur choisit de ne pas exprimer ses préférences, il risque d'aboutir à la situation inverse : un réseau de contraintes possédant un grand nombre de solutions dont certaines quasiment inacceptables. Là encore, la connaissance de préférences aurait permis l'élaboration d'une ou plusieurs solutions« préférées ».

Une première proposition, très générale, a été faite par Freuder via la définition du *problème de satisfaction partielle de contraintes* (Freuder 1989). Il s'agit de résoudre, parmi un ensemble de CSP, le CSP consistant préféré par l'utilisateur : on est ramené à un problème d'optimisation.

Depuis, de nombreux algorithmes et propriétés classiques des CSP ont pu être étendus à différents cadres : CSP flous (ou possibilistes) (Rosenfeld *et al.* 1976; Projet CSPFlex. *et al.* 1992; Schiex 1992; Fargier 1992; Fargier *et al.* 1992; 1993) dans lesquels on cherche à minimiser le degré d'importance de la plus importante des contraintes violées, CSP additifs ou MAX-CSP dans lesquels on cherche à minimiser la somme des degrés d'importance des contraintes violées (Freuder and Wallace 1992; Pinkas 1991; Dupin de Saint Cyr *et al.* 1994), CSP probabilistes dans lesquels on cherche à maximiser la probabilité d'être solution (Fargier and Lang 1993)...

L'ensemble de ces extensions ont, depuis, été analysées dans des cadres algébriques génériques (Schiex *et al.* 1995; Bistarelli *et al.* 1995; 1996). Il apparaît que le cadre des CSP flous/possibilistes est très proche du cadre classique, permettant une extension systématique des propriétés et algorithmes traditionnels. Pour les autres cadres (additifs, probabilistes...), il est nécessaire de définir de nouveaux algorithmes de résolution (Shapiro and Haralick 1981; Freuder and Wallace 1992; Wallace 1995; Verfaillie *et al.* 1996; Larrosa and Meseguer 1996; Larrosa *et al.* 1998; Affane and Bennaceur 1998; Larrosa and Meseguer 1998; Larrosa *et al.* 1999; Schiex 2000).

14.8.3 CSP dynamiques

La seconde catégorie d'extension tente de répondre au problème du traitement de réseaux de contraintes qui ne sont pas figés. Il s'agit alors de maintenir une propriété du réseau lors de modifications successives (généralement l'ajout ou le retrait de contraintes). Après une modification, plutôt que de reprendre la résolution du problème à partir de rien, il semble souhaitable de développer des techniques incrémentales, réutilisant une partie, au moins, du travail déjà effectué sur le problème précédent qui est, en général,

assez semblable. Les problèmes sont assez proches des problèmes traités par les TMS (Truth Maintenance Systems) en logique propositionnelle.

Ce type de problème se rencontre, par exemple, dans le cadre de la mise à jour d'un CSP suite à l'arrivée de nouvelles informations (ordonnancement « réactif »), dans le cadre d'une résolution distribuée (chaque sous-système de résolution devant prendre en compte des contraintes imposées dynamiquement par d'autres sous-systèmes) ou dans le cadre de l'interaction entre un résolveur automatique de CSP et un homme.

Plusieurs approches peuvent être envisagées :

1. maintien d'un certain niveau de consistance locale après chaque modification. Les algorithmes incrémentaux existants traitent uniquement le problème de maintien de la consistance d'arc (Bessière 1992a; Prosser *et al.* 1992; Berlandier and Neveu 1994; Debruyne 1995). Les algorithmes classiques sont incrémentaux pour l'ajout de contraintes, mais le retrait constitue un problème délicat. Les approches existantes sont assez proches des TMS (JTMS) ;
2. maintien d'une solution après chaque modification. Le problème est difficile car l'espace exploré pendant une satisfaction croît exponentiellement dans le pire des cas et il est impossible d'envisager une mémorisation fine pour aboutir à l'incrémentalité. Plusieurs approches, dans des directions variées, ont été proposées (cf. (Minton *et al.* 1992; Verfaillie 1993; van Hentenryck and Provost 1991; Schiex and Verfaillie 1993b)) ;
3. calcul prévisionnel d'une solution *souple i.e.*, de la représentation « compacte » d'une partie ou de toutes les solutions, permettant ainsi de faire face plus rapidement aux aléas (Lesaint 1993; Hubbe and Freuder 1992; Vempaty 1992; Bryant 1986; Brace *et al.* 1990) ;
4. calcul prévisionnel d'une solution *robuste i.e.*, résistant « le mieux possible » aux aléas ;

Les différentes approches ne s'excluent pas. Le lecteur intéressé peut se reporter à (Projet CSPFlex. *et al.* 1992).

14.9 Quelques problèmes

Afin de motiver le lecteur pour l'écriture d'un petit résolveur de contraintes utilisant les techniques précédentes, voici quelques exemples « amusants » de problèmes sous contraintes, modélisables sous la forme de CSP.

Les problèmes de cryptarithmétique ont longtemps été la catégorie la plus représentée. L'instance la plus connue est sans doute « `SEND+MORE =MONEY` ». Rappelons que dans ces problèmes, il faut lier chaque lettre avec une valeur entre 0 et 9, sans que deux lettres aient la même valeur et sans que les chiffres les plus significatifs (*i.e.*, les plus à gauche de chaque « mot » ; `S` et `M` dans ce cas) ne soient nuls :

```
SEND+MORE+GOLD=MONEY    DONALD+GERALD=ROBERT              CROSS+ROADS=DANGER
ADAM+EVE+ON+A=RAFT      AIN+AISNE+DROME+MARNE=SOMME       TWO*TWO=THREE
LYNDON*B=JOHNSON        DEUX+DEUX+DEUX+NEUF+NEUF=FULL     FORTY+TEN+TEN=SIXTY
```

```
ABCD   :   EFG   =   HC
 -          *        +
FHA    -    IG   =   FJD
 =          =        =
AFBE   -   ACFE  =   FAJ
```

Un problème difficile[34] dans le domaine est le suivant : dans la multiplication ci-dessous, chaque point (qui sera représenté par une variable) doit être remplacé par un chiffre pris dans la liste (0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9) (il ne doit pas y avoir plus de deux occurrences du même chiffre, donc précisément deux). Ce problème est présenté dans (Laurière 1976) :

```
   . . .
*    . . .
   -------
     . . .
   . . .
 . . .
---------
. . . . .
```

Citons rapidement, et en vrac (le lecteur intéressé pourra se reporter pour plus de détails et des problèmes plus nombreux à (Roseaux 1986; Gardner 1986; 1987 1988; 1989; Berloquin 1973; Oxusoff and Rauzy 1989; Smullyan 1981; Hofstadter 1990; Laurière 1976; Berge 1970)) :

- **Les N-reines** : placer, sur un échiquier de $N \times N$ cases, N reines qui ne s'attaquent pas (suivant les règles classiques des échecs). Ce problème fut traité (entre autres) par Gauss dans le cas $N = 8$, cas où il oublia quelques solutions... Il s'agit d'un problème équivalent à la recherche d'un ensemble intérieurement stable (qui sera maximum) dans un graphe de $N \times N$ nœuds (un nœud pour chaque case), chaque arête correspondant à une diagonale, une verticale ou une horizontale. Ce problème est, il faut le rappeler, un problème polynomial (voir par exemple (Beasley 1990) pour une procédure de satisfaction polynomiale). Il peut s'envisager pour d'autres types de pièces.
- **Le lemme de Shurr** : il faut disposer m billes numérotées de 1 à m dans n boîtes sans qu'une seule boîte contienne deux billes x et y telles que $x = 2 * y$ ou trois billes telles que $x + y = z$. Habituellement traité avec 3 boîtes. Le problème est consistant jusqu'à $m = 13$;
- **La conjecture d'Erdös** : former une séquence des entiers de 1 à n sans que l'on puisse en extraire une sous-séquence, non forcément contiguë, de k entiers, strictement croissante ou strictement décroissante. En général, on s'intéresse à $k = 4$: le problème est consistant pour $n = 9$, inconsistant pour $n = 10$;
- **Le cavalier d'Euler** : trouver une séquence de mouvement du cavalier permettant de parcourir toutes les cases de l'échiquier, sans repasser par la même case, et en revenant au point de départ ;

34 La résolution devient simple si l'algorithme de satisfaction exploite la dépendance fonctionnelle des quatorze « points » inférieurs par rapport aux six « points » du haut. Encore faut-il qu'il la détecte...

- **Le problème du Zèbre :** attribué à Lewis Carroll, pasteur logicien et écrivain anglais auteur de nombreux autres « puzzles ». On considère cinq maisons, toutes de couleurs différentes (rouge, bleu, jaune, blanc, vert), dans lesquelles logent cinq professionnels (peintre, sculpteur, diplomate, docteur et violoniste) de nationalité différente (anglaise, espagnole, japonaise, norvégienne et italienne) ayant chacun une boisson favorite (thé, jus de fruits, café, lait et vin) et des animaux favoris (chien, escargots, renard, cheval et zèbre). On dispose des faits suivants :
  - l'Anglais habite la maison rouge ;
  - l'Espagnol possède un chien ;
  - le Japonais est peintre ;
  - l'Italien boit du thé ;
  - le Norvégien habite la première maison à gauche ;
  - le propriétaire de la maison verte boit du café ;
  - la maison verte est à droite de la blanche ;
  - le sculpteur élève des escargots ;
  - le diplomate habite la maison jaune ;
  - on boit du lait dans la maison du milieu ;
  - le Norvégien habite à coté de la maison bleue ;
  - le violoniste boit du jus de fruit ;
  - le renard est dans la maison voisine du médecin ;
  - le cheval est à coté de la maison du diplomate.

  Il s'agit de trouver le possesseur du zèbre et le buveur de vin. En fait, le problème n'admet qu'une seule solution et il s'agit de la trouver.
- **Règle de Golomb** : une règle de Golomb est un ensemble d'entiers non négatifs tels que toutes les différences deux à deux dans cet ensemble sont distinctes. Une règle de Golomb à n marques optimale est celle dont l'élément maximum est le plus petit. Ainsi, l'ensemble $(0, 1, 3, 7)$ est une règle de Golomb à 4 éléments puisque les différences deux à deux sont $1 = 1-0, 2 = 3-1, 3 = 3-0, 4 = 7-3, 6 = 7-1, 7 = 7-0$, qui sont toutes distinctes. Une règle à 4 marques optimale est $(0, 1, 4, 6)$ dont les différences deux à deux sont prises dans $(1, 2, 3, 4, 5, 6)$.
  La longueur des règles optimales ne sont connues que pour $n \leq 23$ mais les choses évoluent vite. Ces règles sont utilisées dans différents domaines tels que l'astronomie (placement d'antennes), les détecteurs aux rayons X (placement des capteurs) et de nombreux autres domaines (cryptage des données...).
- **calcul des nombres de Ramsey symétriques** R_p^q : il s'agit du plus petit entier n tel qu'il soit impossible de colorier les arêtes du graphe complet K_n en q couleurs sans faire apparaître une clique de p éléments dont les arêtes soient de la même couleur (cf. (Berge 1987), page 116). Pour $p = q = 3$, il faut donc chercher à colorier les arêtes d'un graphe complet de n sommets en utilisant trois couleurs différentes et de façon à ce qu'aucun triangle ne soit mono-chromatique. Le problème est « facile » à résoudre jusqu'à $n = 10$ sommets, très peu d'algorithmes à vocation générale arrivent à résoudre le problème pour 16 sommets. Le problème est inconsistant pour 17 sommets ou plus (et donc $R_3^3 = 17$). Ce sont les très nombreuses symétries du problème (toute permutation des sommets sur une solution définit une autre solution) qui rendent les algorithmes inefficaces, surtout lorsqu'il s'agit de démontrer l'inconsistance pour $N = 17$. Celle-ci devient beaucoup plus facile dès lors que ces symétries sont détectées par le programme de satisfaction ou

« brisées » par l'utilisateur qui transforme le problème en un problème dont la satisfiabilité est équivalente au problème initial en ajoutant des contraintes différenciant deux sommets distincts (Oxusoff and Rauzy 1989; Benhamou and Sais 1992; Puget 1993). À titre indicatif, on sait que $65 \leq R_3^4 \leq 66$;

- **CSP aléatoires** : une pratique qui devient de plus en plus courante est de tester un nouvel algorithme sur un jeu important de CSP binaires générés entièrement aléatoirement (cf. (Hubbe and Freuder 1992; Selman *et al.* 1992)). Ces tests permettent d'évaluer partiellement les performances relatives de différents algorithmes mais l'uniformité de ces problèmes rend cette mesure peu informative lorsqu'il s'agit, *in fine*, de résoudre des problèmes non-aléatoires et souvent très structurés. On constate expérimentalement que pour presque tous les CSP ainsi générés, le problème de satisfiabilité est très simplement résolu (Cheeseman *et al.* 1991).

CHAPITRE 15

La programmation des jeux

Jean-Marc Alliot — Thomas Schiex

15.1 Introduction

Les jeux, activité futile en apparence, ont toujours fasciné les informaticiens[1]. Ainsi, Babbage[2] avait envisagé de programmer sa machine analytique pour jouer aux échecs, Claude Shannon[3] décrivit en 1950 les rudiments d'un programme d'échecs, comme le fit également Alan Mathison Turing. Dès les débuts de l'IA, les premiers chercheurs se lancèrent dans la réalisation de programme de jeux[4]. Ainsi, un des tous premiers programmes de Simon, Newell, Shaw fut un programme de jeux d'échecs et Arthur Samuel écrivit dans les années 50 un programme de jeu de dames américaines.

Pourquoi cette fascination des informaticiens pour le domaine des jeux ? Tout d'abord parce qu'il semble intuitivement (et inconsciemment) que la pratique de jeux intellectuels soit le propre de l'homme. De plus, tout jeu apparaît un peu comme un champ clos de tournoi au Moyen Âge : chacun est astreint à respecter des règles strictes et précises, et ne peut l'emporter que grâce à sa valeur et à son habileté. La victoire et la défaite sont sanctionnées clairement et sans appel possible. Réaliser une machine qui joue aussi bien qu'un homme, voire mieux, « prouverait » l'intelligence (ou la supériorité) des ordinateurs, et serait apparue comme une grande réussite pour l'IA, du moins à ses débuts.

Enfin, comme le dit Donald Michie :

> « La recherche sur le jeu d'échecs est le champ le plus important de la recherche cognitive[5]. Les échecs seront pour nous ce que la drosophile[6] a été

---

1 Voir (Marsland and Schaeffer 1990) : un ouvrage de référence, à la fois sur les techniques de programmation, et sur les relations de la programmation des jeux et particulièrement des échecs avec l'IA.

2 Charles Babbage (1792–1871) peut être considéré comme un des pères des machines programmables. Il définit dès 1833 les principes fondamentaux qui allaient amener à la réalisation des ordinateurs modernes. Protégé et financé par la comtesse Ada Lovelace, fille de Lord Byron, il ne put cependant mener à bien son projet en raison du manque de moyens technologiques de son époque.

3 Claude Shannon est un des pères de la théorie de l'information.

4 Le petit article (Marsland 1990) de Tony Marsland fait le point sur l'histoire de la programmation du jeu d'échecs.

5 Il est particulièrement intéressant de remarquer qu'aujourd'hui les meilleurs programmes s'appuient sur des méthodes de force brute et non sur des modèles cognitifs. Sur le problème de la force brute dans la programmation des jeux, voir (Michie 1990).

6 La phrase « Les échecs sont la drosophile de l'Intelligence Artificielle » doit être attribuée, d'après John McCarthy, au physicien soviétique Alexandre Kronrod, qui l'utilisa sans doute pour la première fois en 1966. McCarthy a repris l'expression a son compte dans (McCarthy 1990).

pour les généticiens : un moyen simple et pratique de développer de nouvelles techniques. »

Sur le plan technique, le domaine des jeux est très « spécifiable » au sens informatique du terme : les règles sont généralement simples, et surtout les interactions avec le monde extérieur parfaitement nulles : il s'agit d'un *micro-monde* totalement clos, et donc aisément accessible (en apparence) à un programme d'ordinateur. D'autre part, les informaticiens pensaient (plus perfidement) que sa puissance de calcul devrait donner, dans un univers aussi restreint, un avantage considérable à l'ordinateur.

Les choses ne se passèrent pas aussi facilement. Il fallut rapidement distinguer les jeux très simples (morpion) où une analyse exhaustive est possible, des jeux plus complexes comme les dames ou les échecs qui demandent des algorithmes spécifiques et des méthodes heuristiques de recherche, ou encore des jeux à information partielle comme le bridge ou le poker qui demandent en plus des raisonnements de type probabiliste.

Nous allons dans ce chapitre faire un tour d'horizon des techniques de programmation des jeux, et examinerons en détail trois exemples : Othello, les échecs et le bridge. Ces exemples nous permettront d'introduire quelques principes supplémentaires importants de la théorie moderne de la programmation des jeux.

Les joueurs (surtout d'échecs) ayant été des sujets d'études pour les psychologues, nous présenterons quelques-uns des résultats les plus intéressants, résultats qui sont d'autant plus significatifs qu'ils dégagent des facteurs communs à presque toutes les activités humaines.

15.2 Principe minimax (négamax)

Les programmes de jeux à deux joueurs et à information totale utilisent des techniques de représentation semblables à celles que nous avons vues dans le paragraphe précédent : espace d'états et arbres ou graphes. On utilise également des systèmes de production pour générer les états. En revanche, du fait de la présence de deux joueurs ayant des objectifs antagonistes (et non plus identiques), la recherche dans ces arbres[7] ne peut se faire en recourant à des algorithmes du type A^*.

La méthode générale utilisée pour ces jeux est la suivante : à partir d'une position (ou état) donnée, on génère l'ensemble des positions (ou états) que l'ordinateur peut atteindre en jouant un coup (niveau 1 de profondeur). À partir de chacune de ces positions du niveau 1, on génère l'ensemble des positions que l'adversaire peut à son tour atteindre (niveau 2). On peut alors recommencer l'opération aussi longtemps que le permet la puissance de calcul de l'ordinateur et générer les niveaux $3, 4, \ldots, n$. On construit ainsi un arbre de l'espace d'états du problème. Il est clair qu'il est impossible, dans la plupart des jeux, de générer l'ensemble de l'espace d'états du problème. Ainsi, aux échecs, le facteur de branchement[8] est environ de 35 à chaque niveau de profondeur. Une partie d'échecs convenablement jouée devant contenir au moins une trentaine de demi-coups, le nombre d'états de l'espace d'états est alors de l'ordre de

7 Voir (Kaindl 1990) : un excellent article faisant le point sur les méthodes modernes de recherche dans les arbres de jeux.

8 Le facteur de branchement est le nombre de nœuds moyens que l'on peut générer à partir d'une position.

$35^{30} = 2099139642966190174954314623028039932250976562 5$: même l'ordinateur le plus puissant du monde ne pourrait le générer en un temps raisonnable (inférieur à l'âge de l'univers par exemple). On doit donc limiter l'exploration à une *profondeur maximale de résolution*. Lorsqu'on a atteint cette profondeur, l'ordinateur attribue à chacune des feuilles une valeur par l'intermédiaire d'une *fonction d'évaluation*. Cette valeur correspond à l'estimation de la position. Cette estimation peut se faire du point de vue d'un joueur fixe (convention minimax) ou du joueur qui a le trait à cette profondeur (convention négamax). Il doit alors, à partir du niveau 0 (la racine), jouer le coup de niveau 1 qui lui garantit le gain maximal *contre toute défense de son adversaire*, en supposant que celui-ci utilise également une stratégie optimale, c'est-à-dire qu'il joue lui-même à chaque coup le gain qui lui garantit le gain maximal contre toute défense. Ce mécanisme est appelé *principe minimax*[9].

Nous allons détailler les deux phases principales de la résolution : la fonction d'évaluation des états terminaux et les algorithmes de recherche.

15.2.1 Fonctions d'évaluation

La fonction d'évaluation tente d'associer à une position une valeur estimant la qualité de la position pour un des deux joueurs. Les fonctions d'évaluation furent étudiées très tôt. Les premières proposées furent évidemment les plus simples. Aux dames par exemple, on peut effectuer la différence entre le nombre de ses pions et le nombre de pions de l'adversaire. Il est clair qu'une fonction heuristique élaborée permet de mieux jouer : si j'inclus dans ma fonction d'évaluation des notions d'attaque et de défense de pions, j'améliore mon niveau de jeu. En revanche, plus la fonction est élaborée et plus il faut de temps pour la calculer, ce qui limitera la profondeur d'exploration de l'arbre. Nous examinerons plus en détail les fonctions d'évaluation en étudiant la programmation des échecs et d'Othello.

15.2.2 Algorithme minimax

L'algorithme minimax[10] est un algorithme de type *exploration en profondeur d'abord* comme nous les avons décrits au chapitre précédent. Il implante le principe minimax que nous avons évoqué. Le jeu comporte deux joueurs : O (l'ordinateur) et H (son adversaire). Une fonction d'évaluation h est chargée d'évaluer la qualité d'une position de jeu terminale (position de profondeur maximale ou sans descendance) du point de vue de O (par exemple). À chaque niveau où O a le trait, il peut choisir son coup et choisira donc celui de valeur maximale pour lui, on parle de nœud de type Max et du joueur Max. À chaque niveau où H a le trait, O va supposer que l'adversaire essaie de le mettre en mauvaise posture (minimiser ses gains) et choisira donc le coup de valeur minimale pour O (nœud de type Min, joueur Min). Ce n'est que sur les feuilles que l'on peut directement appliquer la fonction d'évaluation h. Sinon, on s'appuie sur une écriture récursive. L'algorithme, qui travaille en profondeur d'abord et jusqu'à la profondeur n, ne stocke jamais plus de n positions à la fois. En effet, il commence par générer une branche complète. Il évalue

9 On peut trouver une présentation complète de cet algorithme avec bien d'autres dans (Adelson-Velsky *et al.* 1952).

10 Les premières descriptions de cet algorithme remontent à la fin des années 40. Ce sont Turing et Shannon qui, les premiers, ont envisagé ce mécanisme de recherche pour les ordinateurs.

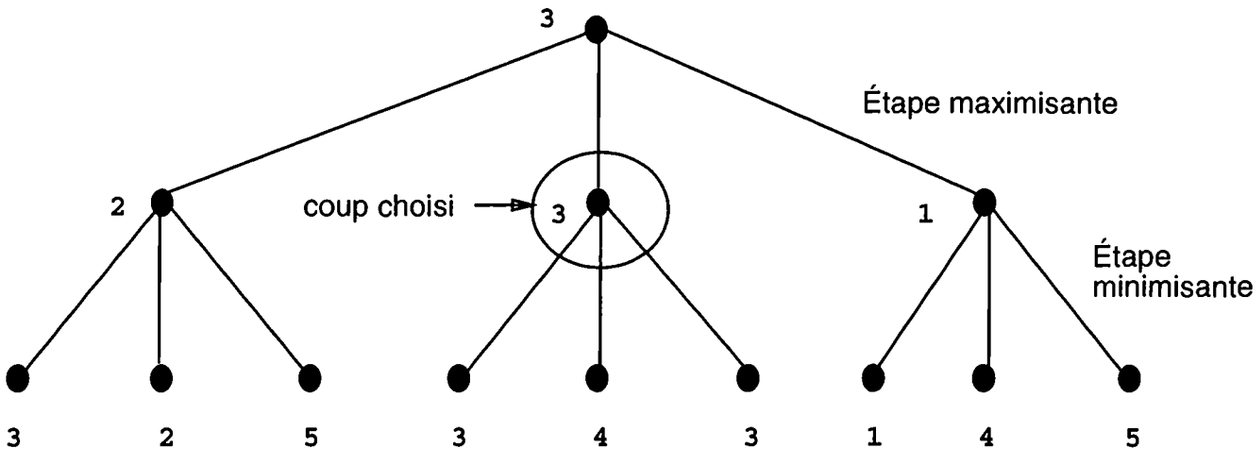


Figure 15.1 – Parcours minimax d'un arbre de jeu

alors la feuille terminale et marque le sommet $n-1$ avec cette valeur. Il repart alors de la position de niveau $n-1$ et génère la feuille suivante (niveau n). Il utilise l'évaluation de cette feuille pour modifier le marquage du niveau $n-1$, et répète l'opération jusqu'à ce qu'il ait généré toutes les feuilles de niveau n issues de cette position de niveau $n-1$. Il utilise alors la valeur de ce niveau $n-1$ pour marquer le niveau $n-2$, remonte à la position du niveau $n-2$ et génère la position de niveau $n-1$ suivante, et continue le processus jusqu'à ce que toutes les feuilles et tous les niveaux aient été générés.

Une convention simplificatrice (du moins pour le programmeur) consiste à considérer qu'on évalue la qualité d'une position non pas du point de vue d'un joueur fixe donné (joueur racine) mais du point de vue du joueur qui a le trait sur cette position. La fonction heuristique doit évidemment être modifiée en conséquence. Dans tous les cas, que ce soit O ou H qui ait le trait, le raisonnement sera le même : pour évaluer le mérite d'un nœud du point de vue du joueur qui a le trait, en connaissant le mérite des positions filles (du point de vue de l'adversaire, qui a le trait sur ces positions), on considère le coup qui maximise (car on veut gagner) l'opposé des mérites des positions filles (car elles ont été évaluées par l'adversaire). Dans tous les cas, la valeur d'un nœud est simplement le maximum des opposés de la valeur des fils. L'écriture de la fonction récursive s'en voit simplifiée. On parle de convention négamax.

Un exemple de parcours par un algorithme minimax est fourni sur la figure 15.1.

On constate que l'algorithme minimax doit complètement décrire l'arbre pour fournir une solution, ce qui, on le verra, est souvent excessif. L'algorithme α-β[11] est une amélioration de l'algorithme minimax qui réalise un élagage de certaines branches qu'il est inutile de visiter.

15.2.3 Algorithme α-β

Examinons l'arbre de la figure 15.2, alors que l'algorithme minimax a déjà parcouru les deux premières branches de la racine, et va s'engager dans la troisième et dernière branche. On sait alors que la racine a une valeur au moins égale à 3 puisqu'un des nœuds fils a déjà été évalué à 3 et que le nœud racine est de type Max. Lorsque, durant l'exploration de la troisième branche, on évalue la première feuille, celle-ci retourne la valeur

11 L'algorithme α-β a été explicitement décrit pour la première fois par Hart et Edwards en 1961.

1. Son père, qui est un nœud Min, ne va pas pouvoir dépasser la valeur 1, ceci *quelles que puissent être les valeurs des deux fils encore inexplorés.* Or, pour modifier la valeur de la racine, et donc le coup à jouer, il faudrait que ce nœud dépasse la valeur 3, puisque la racine est de type Max. Les deux feuilles restantes sont donc inintéressantes et ne seront pas explorées par α-β.

De façon plus synthétique, lorsque dans le parcours de l'arbre de jeu par minimax il y a remise en cause de la valeur d'un nœud, si cette valeur atteint un certain seuil, il devient inutile d'explorer la descendance encore inexplorée de ce nœud.

Il y a en fait deux seuils, appelés pour des raisons historiques, α (pour les nœuds Min) et β (pour les Max) :

- le seuil α, pour un nœud Min n, est égal à la plus grande valeur (connue) de tous les nœuds Max ancêtres de n. Si n atteint une valeur inférieure à α, l'exploration de sa descendance devient inutile ;
- le seuil β, pour un nœud Max n', est égal à la plus petite valeur (connue) de tous les nœuds Min ancêtres de n'. Si n' atteint une valeur supérieure à β, l'exploration de sa descendance devient inutile.

Ceci nous amène à l'énoncé de l'algorithme α-β qui maintient ces deux valeurs durant le parcours de l'arbre (cf. Algorithme 9). Lors de son utilisation, on l'appellera par Alpha-Béta(*Racine*,$-\infty$,$+\infty$).

Algorithme 9: l'α-β (version minimax)

```
;;; n est le sommet dont on veut calculer la valeur;
;;; alpha et β valent initialement −∞ et +∞ respectivement;
Alpha-Béta(n, α, β);
si n est terminal alors retourner h(n);
si n est de type Max alors
    soit j ← 1;
    soit (f_1 ... f_k) les fils de n;
    tant que (α < β et j ≤ k) faire
        α ← max(α,Alpha-Béta(f_j, α, β));
        j ← (j + 1);
    retourner α;
sinon
    soit j ← 1;
    soit (f_1 ... f_k) les fils de n;
    tant que (α < β et j ≤ k) faire
        β ← min(β,Alpha-Béta(f_j, α, β));
        j ← (j + 1);
    retourner β;
```

Une définition beaucoup plus concise, mais peut-être moins intuitive, peut être obtenue en utilisant la convention négamax ; il devient alors inutile de distinguer les nœuds Min des nœuds Max (cf. Algorithme 10). Cet algorithme calcule alors la valeur négamax de la racine qui est, au signe près, égale à sa valeur minimax.

L'algorithme est d'autant plus efficace que l'on parcourt l'arbre « de la bonne façon ». Quand le niveau est maximisant, il faut examiner d'abord les coups qui ont le

Algorithme 10: l'α-β (version négamax)

```
;;; n est le sommet dont on veut calculer la valeur;
;;; alpha et β valent initialement −∞ et +∞ respectivement;
Alpha-Béta(n, α, β);
si n est terminal alors retourner h(n);
sinon
    soit j ← 1;
    soit (f_1 ... f_k) les fils de n;
    tant que (α < β et j ≤ k) faire
        α ← max(α, −Alpha-Béta(f_j, −β, −α));
        j ← (j + 1);
    retourner α;
```

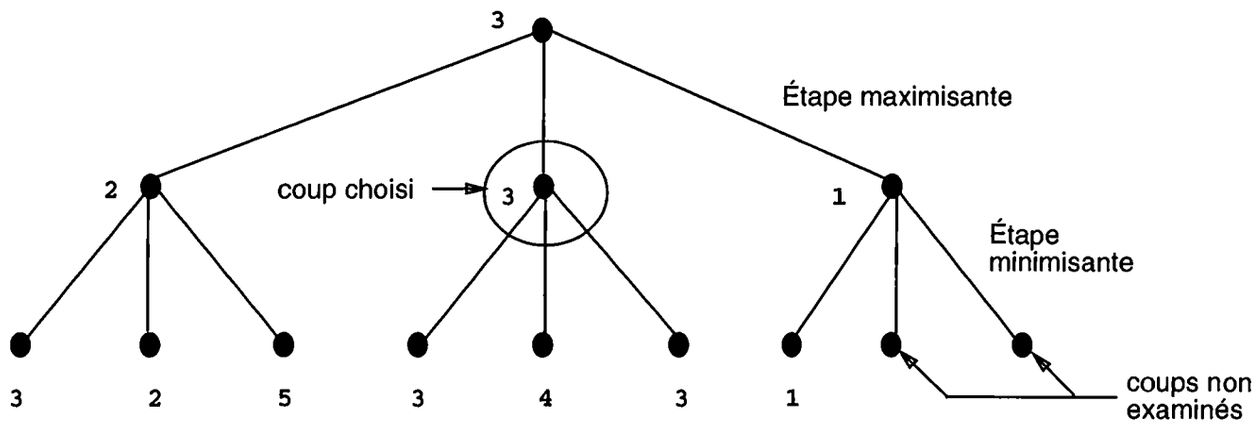


Figure 15.2 – Parcours α-β d'un arbre de jeu

plus de chance de générer des positions à forte valeur d'évaluation et réciproquement. Nous verrons dans la suite de ce chapitre comment réaliser de tels mécanismes. Si l'arbre est parcouru exactement dans le bon ordre, le nombre de feuilles examinées est environ $2\sqrt{N}$ où N est le nombre de feuilles total de l'arbre, alors que l'algorithme minimax examine, lui, les N feuilles.

15.3 L'algorithme SSS*

Il s'agit d'un algorithme relativement peu connu, en tous cas nettement moins connu que l'algorithme α-β. Il a pourtant été démontré qu'il lui est théoriquement supérieur[12], dans le sens où il n'évaluera pas un nœud si α-β ne l'examine pas, tout en élaguant éventuellement quelques branches explorées par α-β. Cette qualité supplémentaire se paie, SSS* étant un gros consommateur de mémoire.

Définition 15.1 – Stratégie – *Étant donné un arbre de jeu $\mathcal{J}$, on appelle* stratégie *ou* sous-arbre solution *pour le joueur* Max, *un sous-arbre de $\mathcal{J}$ qui :*

12 Lors de l'exploration du même arbre, les feuilles étant ordonnées de façon similaire dans les deux cas.

– *contient la racine de $\mathcal{J}$;*
– *dont chaque nœud* Max *a exactement un fils ;*
– *dont chaque nœud* Min *a tous ses fils.*

Une *stratégie* indique au joueur Max ce qu'il doit jouer dans tous les cas. S'il s'agit d'un nœud Max, il a toute liberté de choix et jouera donc le coup indiqué par la stratégie. Si c'est un nœud Min, la stratégie contient tous ses fils et envisage donc toutes les réponses éventuelles de l'adversaire. Si Max respecte une stratégie, il est assuré d'aboutir à une des feuilles de la stratégie. Si l'on se place dans le cas habituel où l'arbre de jeu est développé jusqu'à une profondeur n, le mérite des positions terminales étant estimé par une fonction d'évaluation, la valeur d'une stratégie pour Max est donc le minimum des valeurs des feuilles de cette stratégie, gain assuré contre toute défense de Min. Le but de SSS* est d'exhiber la stratégie de valeur maximum pour Max. Sa valeur sera, par définition, la valeur minimax de l'arbre $\mathcal{J}$.

Définition 15.2 – Stratégie partielle – *Étant donné un arbre de jeu $\mathcal{J}$, on appelle* stratégie partielle *pour le joueur* Max, *un sous-arbre de $\mathcal{J}$ qui :*
– *contient la racine de $\mathcal{J}$;*
– *dont chaque nœud* Max *a au plus un fils.*

Une *stratégie partielle $\mathcal{P}$* représente implicitement l'ensemble des stratégies complètes auxquelles on peut aboutir en développant $\mathcal{P}$ via :
– l'ajout d'un fils à un nœud Max qui n'en a pas. Pour chaque fils, on aboutit à une nouvelle stratégie partielle ;
– l'ajout d'un ou plusieurs fils à un nœud Min, raffinant la stratégie partielle $\mathcal{P}$.

Toutes les stratégies complètes représentées par une stratégie partielle partagent les nœuds de la stratégie partielle, et en particulier, elles partagent les nœuds terminaux sur lesquels on a appliqué la fonction d'évaluation. La valeur d'une stratégie étant égale au minimum de la valeur de ses feuilles, la valeur d'une stratégie partielle constitue une borne supérieure (*i.e.*, une estimation optimiste) de la valeur des stratégies complètes qu'elle représente.

L'algorithme SSS* explore un espace d'état dont chaque nœud est une stratégie partielle, en utilisant une approche *meilleur d'abord* avec une heuristique minorante qui sera la valeur des stratégies partielles et qui garantit l'optimalité. Il ajoute un raffinement permettant d'éviter le développement de certaines stratégies partielles dont on peut démontrer l'absence d'intérêt.

Considérons par exemple l'arbre de jeux de la figure 15.3 et appliquons un *meilleur d'abord* avec une heuristique égale à la valeur des stratégies partielles (valeur de $+\infty$ si la stratégie ne contient pas de nœud terminal). Dans la suite, nous décrirons une stratégie partielle par un couple formé de l'ensemble des nœuds qu'elle contient et de sa valeur estimée. Nous désignerons par G la liste des stratégies partielles.

1. $G = ((\{1\}, +\infty))$. Il y a deux façons de développer la stratégie $(\{1\}, +\infty)$: en ajoutant le nœud 2 ou le nœud 3. Cependant, toutes les stratégies complètes issues de 1 contiennent les nœuds 2 et 3 car le nœud 1 est de type Min. Afin d'éviter d'explorer deux voies menant à la même solution, nous commencerons systématiquement par le premier nœud dans l'ordre lexicographique.
2. $G = ((\{1, 2\}, +\infty))$. Le nœud 2 étant de type Max, le rajout des nœuds 4 ou 5 définit deux stratégies partielles distinctes qu'il nous faut envisager.

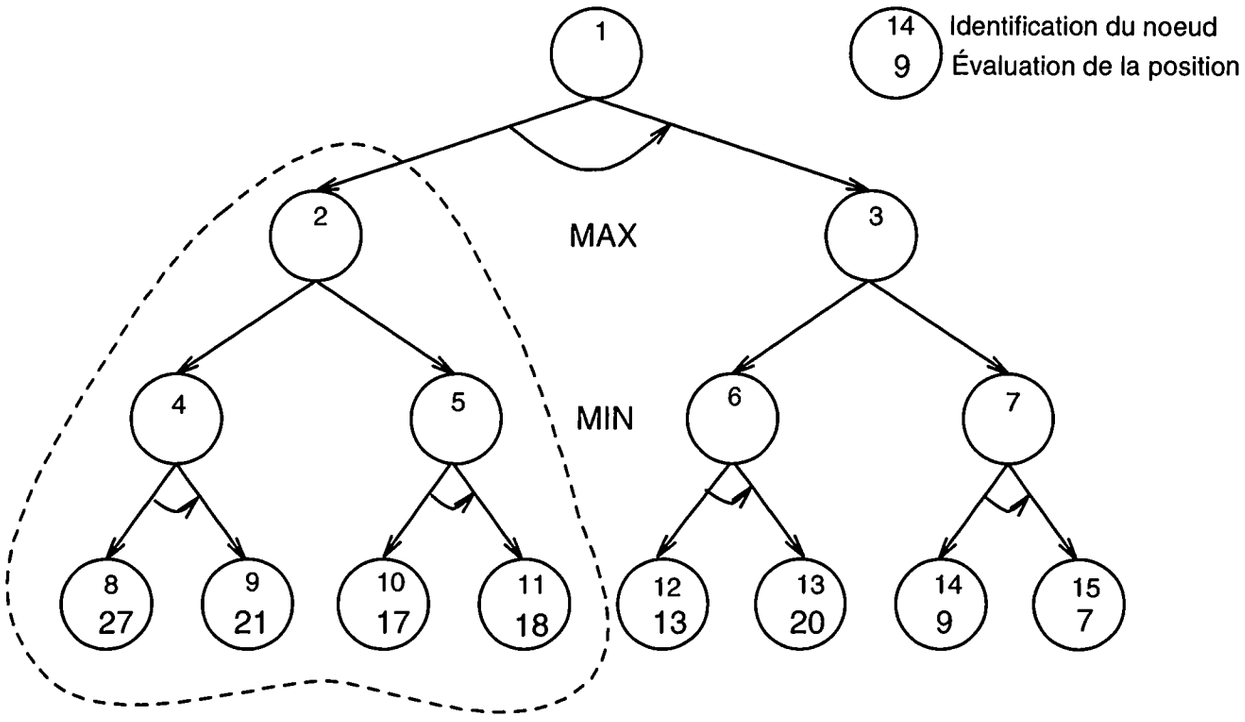


Figure 15.3 – SSS* et arbre de jeu

3. $G = (\{1,2,4\},+\infty)(\{1,2,5\},+\infty))$. On développe la première stratégie partielle.
4. $G = (\{1,2,5\},+\infty)(\{1,2,4,8\},27))$. La stratégie $(\{1,2,5\},+\infty)$ est passée en tête.
5. $G = (\{1,2,4,8\},27)(\{1,2,5,10\},17))$. Il faut maintenant affiner l'estimation de la stratégie partielle $(\{1,2,4,8\},27)$. Les différents nœuds envisageables sont 9 et 3. On choisit 9.
6. $G = (\{1,2,4,8,9\},21)(\{1,2,5,10\},17))$. À ce point du développement, il faut noter que la stratégie complète optimale issue de 2 est *connue*. Il s'agit de $(\{1,2,4,8,9\},21)$. On étend la première stratégie avec 3.
7. $G = (\{1,2,4,8,9,3\},21)(\{1,2,5,10\},17))$. Le nœud 3 est un nœud Max, il nous faut considérer les deux nœuds 6 et 7.
8. $G = (\{1,2,4,8,9,3,6\},21)(\{1,2,4,8,9,3,7\},21)(\{1,2,5,10\},17))$. Puis, on développe 12.
9. $G = (\{1,2,4,8,9,3,7\},21)(\{1,2,5,10\},17)(\{1,2,4,8,9,3,6,12\},13))$. On passe à 14.
10. $G = (\{1,2,5,10\},17)(\{1,2,4,8,9,3,6,12\},13)(\{1,2,4,8,9,3,7,14\},9))$. La stratégie $(\{1,2,5,10\},17)$ est alors en tête. Or, elle contient la sous-stratégie $(2,5,10\},17)$ dont on sait, d'après la remarque faite au (6) qu'elle est sous-optimale. Il est donc *inutile* de l'explorer plus avant.

L'algorithme du SSS* s'inspire de cette remarque. À partir du moment où une sous-stratégie optimale est établie (comme au point (6)), la stratégie optimale est marquée, et toutes les sous-stratégies sous-optimales supprimées de G. En voici la description précise. Nous noterons :

1. $f_1(n)$: le premier fils du nœud n ;

2. $f_i(n)$: un fils de n ;
3. $\mathrm{fr}(n)$: le frère suivant de n (quand il existe) ;
4. $p(n)$: le père de n.

On dira qu'un nœud est *résolu*, si la stratégie complète issue de ce nœud a été déterminée. Un nœud qui n'est pas résolu est *vivant*. Il ne faut pas confondre par la suite nœud (lié à l'arbre de jeu) et état (lié à l'arbre d'exploration des stratégies).

- Un état sera donc un triplet de la forme *(nœud, type, valeur)* où *nœud* dénote un nœud de l'arbre de jeu, *type* vaut *résolu* ou *vivant* et *valeur* est la valeur estimée de la stratégie partielle que l'on vient d'étendre en ajoutant le *nœud*.
- La liste G des états générés non développés sera, tout comme dans l'algorithme *meilleur d'abord*, triée par ordre décroissant des valeurs. Cette liste sera manipulée par les opérateurs Premier, ExtraitPremier et Inserer qui permettent respectivement d'accéder au meilleur état, d'extraire le meilleur état et d'insérer un nouvel état en maintenant la liste ordonnée[13].
- $h(n)$, retournera la valeur estimée d'un nœud de l'arbre de jeu du point de vue de l'un des deux joueurs.

Algorithme 11: Le SSS*.

```
;;; n_0 est le nœud à évaluer.
G ← ((n_0, vivant, +∞));
tant que Premier(G) n'est pas de la forme (n_0, résolu, v) faire
    (n, t, e) ← ExtraitPremier(G);
    si t = vivant alors
        suivant le type de n faire
1           cas où n est terminal : Inserer((n, résolu, min(e, h(n))), G);
            cas où n est Max :
                soit (f_1 ... f_k) les fils de n;
2               pour i allant de 1 à k faire Inserer((f_i(n), vivant, e), G);
3           cas où n est Min : Inserer((f_1(n), vivant, e), G)
    sinon
        si n est de type Min alors
            Inserer((p(n), résolu, e), G);
4           Supprimer de G tous les états dont le nœud est un successeur de p(n);
        else
5           si n a un frère suivant alors Inserer((fr(n), vivant, e), G);
6           sinon Inserer((p(n), résolu, e), G);
retourner v ;
```

L'algorithme (cf. Algorithme 11) traite le meilleur état courant (n, t, e) selon son type. Si l'état est marqué vivant, cela signifie que la meilleure stratégie issue du nœud correspondante$n$ n'est pas encore encore connue et il faut continuer à la developper :

13 Ces fonctionnalités peuvent être obtenues par une structure de données du type « file de priorité » ou *Heap* (Cormen *et al.* 1990).

- Si le nœud est terminal (ligne 1) il suffit d'appliquer l'heuristique et l'état est marqué *résolu.*
- Si le nœud est de type Max, chacun des fils de n défini une stratégie partielle et il faut donc insérer un nouvel état pour chacun d'entre eux (ligne 2) ;
- Si le nœud est de type Min, une stratégie complète devra contenir tous les fils de n. On commence par étendre la stratégie courante avec le premier fils de n (ligne 3). Il faudra ultérieurement considérer ses frères (ligne 5).

Si l'état (n, t, e) est marqué résolu, une stratégie complète optimale issue du nœud est connue et il faut faire remonter l'information :

- Si le nœud est de type Min (fils d'un nœud Max), on est revenu sur un état créé à la ligne 2 et une stratégie optimale pour le nœud père est connue. Il faut effacer tous les stratégies concurrentes issues des frères de n (ligne 4) ;
- Si le nœud est de type Max(fils d'un nœud Min), on est revenu sur une un état créé à la ligne 3. Il faut maintenant étendre la stratégie avec le fils suivant du père de n s'il existe (ligne 5). Sinon, le nœud est résolu car tous les fils ont été examinés (ligne 6).

Comme nous l'annoncions initialement, l'algorithme du SSS* est supérieur à α-β[14]. Récemment, un algorithme paramétrable sur la quantité de mémoire (longueur maximum de la liste G) qu'il peut utiliser — l'IterSSS* — a été exhibé (Bhattacharya and Bagchi 1986). Il permet d'obtenir des performances variant entre celles d'α-β (mémoire minimum) et de SSS* (mémoire suffisamment importante). Le tableau suivant indique les performances de cet algorithme en pourcentage de nœuds visités sur différents arbres uniformes de facteurs de branchements et de profondeurs variables :

Profondeur	Branchement	α-β	SSS*	longueur G	IterSSS*
15	2	12,47	8,5	9	12,4
				64	10,36
				192	9,37
				256	8,5
6	5	16,51	11,48	13	15,49
				63	12,68
				95	12,6
				125	11,48
10	3	10,44	6,44	11	10,11
				122	7,75
				243	6,44

14 Il faut pour cela que les deux algorithmes développent le même arbre, dans un ordre comparable. Ceci nécessite, dans le cas de l'insertion dans G de deux nœuds de même valeur, de mettre en première position le nœud exploré en premier par α-β et entraîne l'emploi de finesse de programmation (ordonnancement fixe des nœuds...) qu'il est inutile de pratiquer lorsqu'on désire employer SSS* pour autre chose qu'une comparaison à α-β.

15.4 L'algorithme Scout

Nous décrirons rapidement l'algorithme Scout, élaboré par J. Pearl (Pearl 1990) comme outil théorique. Son efficacité est, en général, inférieure à celle d'α-β, pour une consommation mémoire du même ordre. Il peut toutefois lui être supérieur.

Scout repose sur une idée fort simple : si l'on disposait d'un moyen efficace pour comparer (sans nécessairement la déterminer) la valeur minimax d'un nœud à une valeur donnée, une quantité importante de recherche pourrait être évitée. Considérons par exemple un nœud Max n, ayant deux fils : f_1, dont la valeur v est connue, et f_2. Si l'on sait que la valeur de f_2 est inférieure à v, il est inutile d'explorer la branche de f_2.

Scout s'appuie donc sur deux procédures simples : la première, appelée Test permet de vérifier si la valeur d'un nœud n est strictement supérieure (ou supérieure ou égale) à une valeur donnée v. Nous continuerons à désigner par h la fonction heuristique.

Procedure Test(n, v, c)

```
;;; n est le nœud à évaluer;
;;; v la valeur seuil;
;;; c le prédicat fonctionnel de comparaison (> ou ≥);
si n est un nœud terminal alors retourner c(h(n), v);
si n est Max alors
    soit S = {s_1 ... s_i} la liste des fils de n;
    pour j allant de 1 à i faire
        si Test(s_j, v, c) alors retourner vrai;
    retourner faux;
sinon
    soit S = {s_1 ... s_i} la liste des fils de n;
    pour j allant de 1 à i faire
        si ¬Test(s_j, v, c) alors retourner faux;
    retourner vrai;
```

La seconde, Eval(), utilise Test et applique le principe donné plus haut pour calculer la valeur minimax d'un arbre de jeu. Elle prend en paramètre un nœud n.

Il pourrait sembler que, du fait de la redondance éventuelle des évaluations, lorsque Test ne permet pas la coupure, Scout devrait être très inférieur à α-β, voire même à minimax. Une étude mathématique du comportement asymptotique de Scout montre un comportement identique à α-β pour des profondeurs élevées. Dans le cas de certains jeux (jeu de Kalah), il peut même lui être supérieur (jusqu'à 40% d'amélioration à profondeur 5). Bien qu'aucune preuve ne vienne l'appuyer, il semble que Scout soit plus particulièrement adapté aux arbres profonds, ayant un facteur de branchement faible[15].

Le tableau suivant reprend l'ensemble des algorithmes de jeu présentés (minimax, α-β, SSS*, Scout) et indique les performances relatives de chacun d'eux,

---

15 Le lecteur intéressé pourra l'appliquer à l'Awale, proche du jeu de Kalah, jeu africain dont le facteur de branchement est inférieur à 6.

Procedure Eval(n)

;;; n est le nœud à évaluer;

si n *est un neud terminal* **alors retourner** $h(n)$;
soit $S = \{s_1 \dots s_i\}$ *la liste des fils de* n;
soit $v = Eval((s_1))$;
pour j *allant de* 2 *à* i **faire**
  si s_j *est Max* **alors**
    si $Test(s_j, v, >)$ **alors** $v \leftarrow \mathsf{Eval}(s_j)$;
  sinon
    si $\neg Test(s_j, v, \geq)$ **alors** $v \leftarrow \mathsf{Eval}(s_j)$;
retourner v;

– en nombre de feuilles explorées ;
– et en temps machine consommé[16].
sur le jeu de l'Othello pour différentes profondeurs p.

p	minimax	Scout	α-β	SSS*
1	3 – 0,37	4 – 0,45	3 – 0,36	3 – 0,36
2	14 – 1,38	21 – 2.09	13 – 1,4	11 – 1,19
3	61 – 6,0	45 – 5,0	37 – 3,9	35 – 3,7
4	349 – 32,5	246 – 23,7	150 – 14,77	95 – 10,0
5	2050 – 185,7	615 – 61,6	418 – 41,4	292 – 19,2
6	13773 – 1213,5	2680 – 254,5	1830 – 172,9	1617 – 117,3

On notera les mauvaises performances de Scout à faible profondeur (plus mauvais que le simple minimax) et la nette supériorité de SSS*. Il faut cependant noter que nous n'avons pu poursuivre les tests avec SSS* au-delà de la profondeur 6 pour des raisons de place mémoire.

15.5 La pratique après la théorie

Les différents algorithmes et résultats présentés ci-dessus sont parfaitement intéressants mais ne sont plus utilisés aujourd'hui sous cette forme. De nombreuses améliorations ont été apportées et il est indispensable de les connaitre pour espérer écrire un programme performant.

15.5.1 Contrôle du temps et α-β

On doit dans la pratique s'intéresser au système de contrôle du temps. Le programme dispose d'un temps total pour jouer l'ensemble de la partie. Il serait possible

16 Algorithmes tous réalisés en LE_LISP, utilisant la même fonction heuristique, les mêmes fonctions d'exploration de l'arbre de jeu. Les temps sont sans unité.

position	évaluation	meilleur coup	distance à la position terminale

Figure 15.4 – Informations stockées dans une table de transposition

d'implanter deux techniques pour que l'ordinateur ne consomme pas trop de temps sur un coup. On pourrait tout d'abord accorder un temps donné pour jouer le coup et estimer la profondeur maximale que l'on peut atteindre sans dépasser ce temps. Cette solution présente un inconvénient : il est très difficile de trouver une fonction qui estime le temps que prendra un calcul à une profondeur donnée. On risque de se retrouver en dépassement de temps. On préfère donc utiliser une méthode un peu différente. On fait fonctionner l'algorithme α-β à un niveau minimal (3 par exemple). On compare le temps utilisé avec le temps disponible. S'il reste du temps, on relance la recherche au niveau 4, puis au niveau 5... Cette méthode (habituellement référencée sous le nom de *Depth First Iterative Deepening* (Korf 1985; Slate and Atkin 1978; Marsland and Schaeffer 1990)) présente un autre avantage. On constate aisément qu'un algorithme α-β est d'autant plus efficace qu'il évalue en premier les meilleurs coups. On profite donc du résultat de la recherche à la profondeur n pour classer les coups et faire examiner en premier au programme les meilleurs coups de la profondeur n lorsqu'il effectue la recherche en profondeur $n + 1$.

15.5.2 Tables de transposition

Le principe des tables de transposition est le suivant : pour chaque position rencontrée dans la recherche, l'algorithme minimax retourne une évaluation. Il s'agit de conserver dans une table la valeur de chacune des positions rencontrées de façon à pouvoir réutiliser directement cette valeur lors des recherches suivantes. Les informations stockées pour une position sont représentées sur la figure 15.4. Nous allons voir cela sur un exemple pratique. Soit l'arbre minimax représenté sur la figure 15.5. Les informations stockées pour la position P_1 sont représentées figure 15.6. Supposons que dans une recherche future l'ordinateur rencontre à nouveau la position P_1. Si la distance à la racine de P_1 est plus grande dans la table de transposition que dans la recherche actuelle, l'ordinateur ne poursuivra pas l'évaluation : il se contentera de recopier la valeur stockée dans la table de transposition. En revanche, si la distance à la racine est plus petite dans la table de transposition que dans l'évaluation courante, le programme terminera normalement l'évaluation de la position et remplacera l'ancienne évaluation de la table de transposition par celle qu'il aura calculée, puis stockera la nouvelle valeur de la distance à la racine.

Si les tables de transposition fonctionnaient suivant ce principe, un problème se poserait rapidement : comment stocker tous les éléments de la table de transposition (et en particulier la position de chacune des pièces sur l'échiquier) dans la mémoire du calculateur, et comment retrouver rapidement une position. On utilise une structure de données connue sous le nom de *table de hachage*[17] ; au lieu d'utiliser la position en entier pour retrouver la valeur de cette position, on utilise un nombre qui caractérise cette position.

17 En anglais, *hash table*. Voir par exemple (Cormen *et al.* 1990).

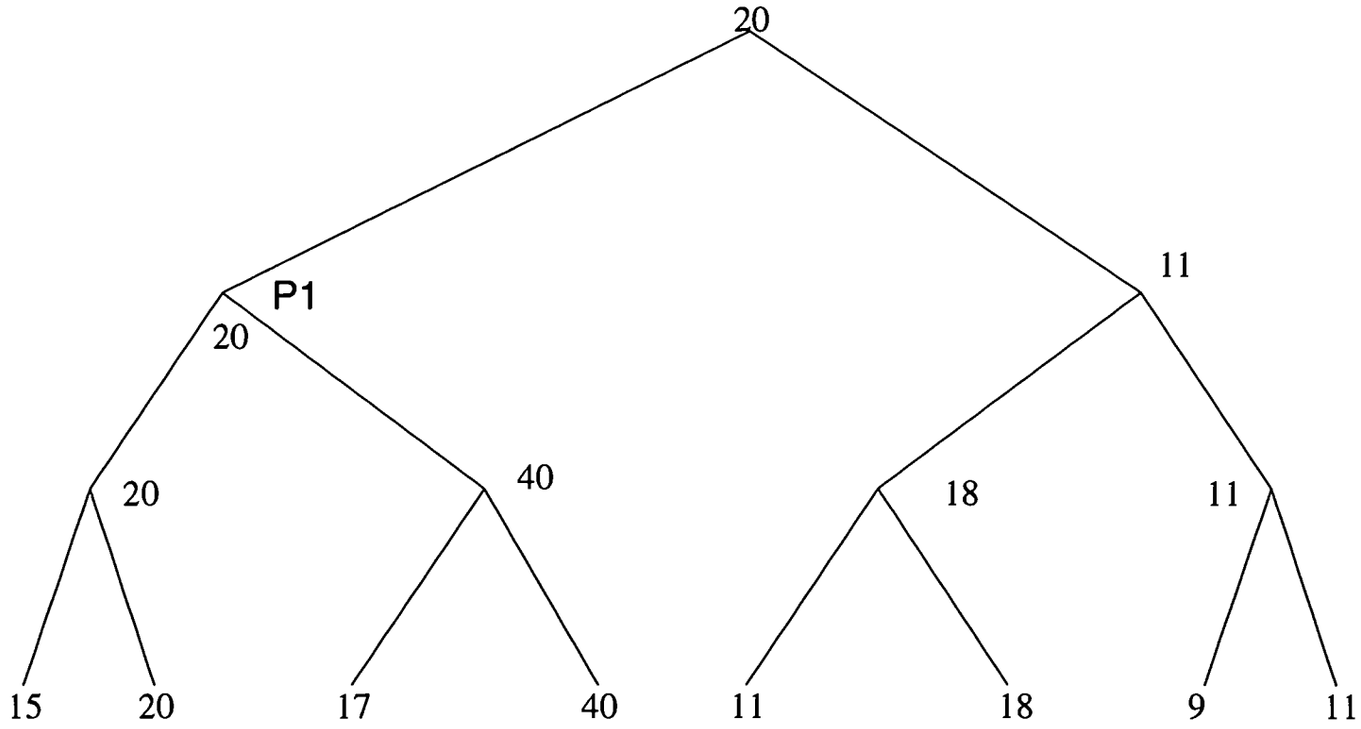


Figure 15.5 – Exemple d'arbre minimax

position	évaluation	meilleur coup	distance à la position terminale
P_1	20	Gauche	2

Figure 15.6 – Valeurs stockées pour la position P_1 de l'arbre minimax représenté sur la figure précédente

Nous allons examiner brièvement sur un exemple une méthode permettant de construire ce nombre pour une position aux échecs.

On construit une table contenant, pour chaque pièce de chaque couleur sur chacune des cases de l'échiquier, un nombre binaire de 32 chiffres choisi de façon aléatoire (voir exemple table 15.1). Pour une position donnée, on construit alors le code de hachage en faisant un « ou-exclusif[18] » entre chacune des valeurs associée à chacune des pièces. Ainsi, pour la position représentée sur la figure 15.7, la valeur sera :

Tour blanche en a1	10101011110110110101010110101101	$\oplus$
Roi blanc en e1	11010110101001001010001011010110	$\oplus$
Roi noir en e8	00100101010110110100100010101000	$=$
	01011000001001001011111111010011	

Le grand intérêt de cette méthode est qu'elle permet de calculer la valeur de hachage d'une position après déplacement d'une pièce, simplement en ajoutant à la valeur de hachage de la position originale le code de la pièce sur la nouvelle case et le code de la pièce sur l'ancienne case. C'est une propriété du « ou-exclusif ».

18 L'addition de nombres binaires sans retenue (qui correspond à un *ou exclusif* bit à bit) consiste à additionner chaque chiffre en utilisant comme règle : $0 \oplus 0 = 0$, $0 \oplus 1 = 1$ et $1 \oplus 1 = 0$.

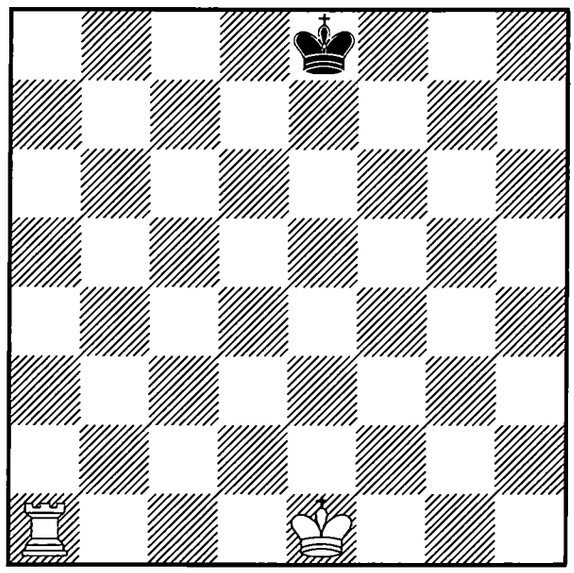


Figure 15.7 – Position

Pièce	couleur	position	valeur de hash-coding (choisie aléatoirement)
Tour	blanc	a1	10101011110110110101010110101101
Tour	blanc	a2	01000101001011001110100111101010
...	...	...	...
Roi	blanc	e1	11010110101001001010001011010110
...	...	...	...
Roi	noir	e8	00100101010110110100100010101000
...	...	...	...

Table 15.1 – Valeurs de hash-coding

Un des problèmes des tables de hachage est que rien ne nous garantit que deux positions différentes ne vont pas avoir la même valeur de hachage, ce qui risquerait de nous amener à des conclusions fausses sur la valeur d'une position. Cette probabilité de collision est d'autant plus faible que la taille de la table est importante.

Il reste maintenant à stocker un élément de la table de hachage. On ne peut utiliser la valeur de hachage comme index de tableau car elle est généralement trop grande (un tableau avec un index sur 32 bits occuperait quatre milliards de positions mémoire, ce qui est déraisonnable). On peut par exemple utiliser les 16 premiers bits de la valeur de hachage comme index du tableau. On prend alors le risque que des positions aient des valeurs de hachage différentes et le même index. Il y a alors *conflit*. Il existe différentes méthodes pour gérer ce type de problème, la plus simple étant de remplacer la plus ancienne des deux valeurs par la plus récente[19].

15.5.3 α-β avec mémoire

La méthode des tables de transposition permet d'améliorer de façon spectaculaire la vitesse de recherche d'un calculateur, car elles permettent l'implantation de la version la

19 On appelle cette structure *hash cache table* en anglais. Notons qu'il existe des stratégies plus évoluées.

plus classiquement utilisée de l'α-β, l'α-β avec mémoire. Sa structure est un petit peu plus complexe que celle de l'α-β standard, car il fait appel aux tables de transposition pour stocker les bornes supérieures et inférieurs des positions déjà évaluées. Nous le présentons ici dans sa version la plus aisée à comprendre (algorithme 14).

On peut aisément se demander quel peut-être l'intérêt d'un tel algorithme. Il est de deux ordres. D'une part, dans les jeux où certaines positions se présentent plusieurs fois dans l'arbre, il permet de ne pas recalculer le nœud comme indiqué précédemment. Mais son principal avantage apparait lorsqu'on l'utilise avec des fenêtres de recherche réduite, en particulier avec les algorithmes de la famille MTD(f).

Algorithme 14: l'α-β (version minimax avec mémoire)

```
;;; n est le sommet dont on veut calculer la valeur;
Alpha-Béta(n, α, β);
si n est dans la table alors
    si n.bas >= β alors retourner n.bas;
    si n.haut <= α alors retourner n.haut;
    α ← max(α, n.bas);
    β ← min(β, n.haut);
si n est terminal alors retourner h(n);
si n est de type Max alors
    soit g ← −∞;
    soit j ← 1;
    soit (f_1 ... f_k) les fils de n;
    tant que (g < β et j ≤ k) faire
        g ← max(g,Alpha-Béta(f_j, α, β));
        α ← max(α, g);
        j ← (j + 1);
sinon
    soit g ← +∞;
    soit j ← 1;
    soit (f_1 ... f_k) les fils de n;
    tant que (g > α et j ≤ k) faire
        g ← min(g,Alpha-Béta(f_j, α, β));
        β ← min(β, g);
        j ← (j + 1);
si g <= α alors n.bas ← g;
si g >= β alors n.haut ← g;
si g > α et g < β alors
    n.bas ← g;
    n.haut ← g;
retourner g
```

15.5.4 Fenêtre réduite et MTD(f)

On se rappelle que lors du début de l'algorithme α-β, les bornes α et β sont initialisées à $+\infty$ et $-\infty$. Or, dans 90% des positions, il est rare que la valeur d'une position évolue beaucoup. On peut donc initialiser les bornes α et β à la valeur de la position courante $\pm\epsilon$ (ou ϵ est une valeur bien choisie dépendant du jeu et/ou de la fonction d'évaluation). Ainsi, tous les coups dans l'algorithme α-β qui seront en dehors de cette fenêtre seront immédiatement élagués. La méthode est efficace si l'hypothèse initiale est juste (c'est-à-dire qu'on ne peut effectivement pas gagner ou perdre plus de ϵ). Si tel n'est pas le cas, on le détecte aisément car la valeur retournée est alors hors de la fenêtre initiale $]\alpha\ ,\ \beta[$. Il faut alors relancer l'algorithme en modifiant la fenêtre de recherche ; mais si l'on s'est donné la peine de stocker dans la table de transposition pour chaque position déjà exploré les informations acquises lors de la passe précédente on gagne un temps précieux en ne réexaminant pas certains nœuds.

C'est un poussant ce raisonnement à l'extrême, et en s'inspirant de l'algorithme Scout, qu'est né l'algorithme MTD(f) (algorithme 15). Cet algorithme présente de

Algorithme 15: MTD(f)

```
;;; root est la racine de l'arbre;
MTD(root, f);
soit g ← f;
soit h ← +∞;
soit b ← −∞;
tant que h > b faire
    si g = b alors β ← g + 1 sinon β ← g;
    g ←Alpha-Béta(root, β − 1, β);
    si g < β alors h ← g sinon b ← g
retourner g
```

nombreuses particularités qui le rendent particulièrement intéressant. Il emprunte en effet certains traits à la plupart des algorithmes que nous avons déjà rencontrés. Son principe est simple : il appelle autant de fois que nécessaire un α-β pour construire un encadrement de la valeur de la position. Quand les deux bornes haute et basse se rencontrent, l'évaluation est terminée et la valeur de la position est connue. L'efficacité de l'algorithme vient de l'élagage violent qu'il effectue à chaque évaluation, car l'α-β avec mémoire est appelé avec une fenêtre de taille 0. Il est d'autant plus efficace que la valeur du paramètre f dont dépend MTD(f) est proche de la valeur réelle de la position. On utilise donc en général comme valeur de f la valeur retourné par l'algorithme lors d'une itération précédente. Remarquons que la technique consistant à utiliser un α-β avec une fenêtre de taille 0 est équivalent à la procédure TEST de l'algorithme SCOUT.

De récents travaux (Plaat *et al.* 1996) ont montré d'autre part que MTD($+\infty$) était exactement équivalent à SSS^*, c'est à dire qu'il développe les mêmes nœuds dans le même ordre, répondant en cela aux nombreuses questions qui se posaient depuis l'article fondateur de SSS^*. On en déduit en particulier que SSS^* est moins efficace que MTD(f) pour une valeur de f bien choisie.

15.5.5 Paralléliser un algorithme α-β

Comme pour tous les programmes utilisant des techniques de recherche dans des arbres, la puissance de calcul est un facteur déterminant. Un moyen traditionnel d'augmenter cette puissance consiste à tenter de rendre l'exécution du programme parallèle, c'est-à-dire d'utiliser plusieurs machines (ou plusieurs processeurs) pour exécuter simultanément des parties du programme.

Il est clair que l'on peut aisément paralléliser un algorithme α-β. En effet, il suffit de distribuer la recherche dans l'arbre sur l'ensemble des processeurs. Mais on ne peut pas le faire n'importe comment. On doit choisir entre quatre types de technique (nous retrouverons ce type de mécanisme lorsque nous parlerons de parallélisme en programmation logique) :

Système distribué contre système en étoile : Dans un système en étoile[20] un des processeurs, le processeur maître, est chargé de diriger la résolution. Ainsi, dans le cas qui nous intéresse, le processeur maître envoie à chacune des machines esclaves une position et lui demande de renvoyer sa valuation α-β. Il peut, s'il le désire, interrompre le travail d'un processeur s'il estime ce travail inutile, à la suite d'informations qu'il aurait reçues d'autres processeurs. Ce type de mécanisme est simple à mettre en œuvre, car le contrôle de l'exécution reste séquentiel, toutes les des décisions étant prises par un seul processeur. D'autre part, le nombre d'informations transitant entre les processeurs reste relativement faible.

Dans un modèle totalement distribué, aucun processeur n'a de statut particulier. Chaque processeur choisit au départ une position à évaluer (au hasard et après un délai aléatoire différent de façon à limiter les conflits) et diffuse sur le réseau à l'intention des autres son choix. Au cours de la résolution, il diffuse également, à l'intention de tous les autres processeurs, l'ensemble des informations qui lui paraissent intéressantes (valuations α-β de position par exemple). Ce type de modèle est plus fiable dans la mesure où la défaillance (ou l'absence) d'une machine est plus facilement surmontée. En revanche, il est beaucoup plus complexe à réaliser. En particulier, la gestion des conflits doit être faite avec soin : comment faire si deux processeurs se rendent compte qu'ils ont choisi la même branche ? Une méthode envisageable consiste à faire attendre chaque machine pendant une durée aléatoire, avant de leur faire choisir à nouveau une branche à évaluer. Enfin, un autre inconvénient est l'augmentation du débit des données qui va circuler sur le réseau de communications entre processeurs.

Grosse ou petite granularité : [21] Lorsque l'on distribue la résolution dans l'algorithme α-β, on doit choisir le niveau à partir duquel chaque processeur va commencer une évaluation α-β séquentielle classique. Si nous choisissons de faire commencer l'évaluation au niveau 1 de l'arbre (grosse granularité), nous ne pourrons pas utiliser plus de processeurs qu'il n'y a de positions différentes au niveau 1 (facteur de branchement). Ainsi, pour Othello, il y a en moyenne 16 coups jouables

---

20 Appelé aussi modèle maître-esclaves.

21 Le terme de granularité semble adapté à la représentation intuitive que l'on se fait de la parallélisation d'une tâche. On peut considérer la tâche comme un bloc monolithique. La paralléliser va consister à la réduire en sous-tâches plus petites. Si nous découpons la tâche en gros blocs, on parlera de grosse granularité (peu de sous-tâches, et chaque sous-tâche est importante). Si nous décomposons la tâche en de multiples petites sous-tâches, on parlera de petite granularité.

à chaque tour. Si nous disposons de n processeurs avec $n > 16$, nous aurons en moyenne $n - 16$ processeurs inemployés. Si nous distribuons la recherche à partir du niveau 2 (256 positions), nous serons en mesure d'occuper simultanément 256 processeurs. En revanche, le nombre d'informations qui circuleront sur le réseau sera beaucoup plus important et continuera à augmenter au fur et à mesure que nous rendrons la granularité plus petite.

Il faut donc choisir sa stratégie en fonction des capacités des machines et du réseau dont on dispose. Examinons en détail le fonctionnement d'un système à forte granularité :

1. Au départ, la machine maître est la seule à connaître la position à évaluer (niveau 0). À partir de cette position, elle établit les 15 premières positions que l'on peut générer au niveau 1 et demande à chacun des processeurs esclaves de calculer la valuation α-β pour une de ces positions.
2. Chaque processeur esclave va, au fur et à mesure que l'évaluation de la position que lui a confiée la machine progresse, renvoyer à la machine maître les évaluations partielles au niveau 1. La machine maître note chacune des évaluations partielles pour chacun des processeurs, en sachant qu'une évaluation partielle de niveau 1 ne peut que diminuer (principe minimax).
3. À un instant donné, une des machines esclaves termine la résolution de sa position et renvoie à la machine maître l'évaluation définitive de cette position de niveau 1. La machine maître effectue alors la séquence d'opérations suivante :
   (a) Elle note ce coup comme le meilleur connu et cette évaluation comme la meilleure évaluation courante et l'appelle p_0.
   (b) Elle note que ce processeur est disponible.
   (c) Elle ordonne à tous les processeurs dont l'évaluation partielle de niveau 1 est inférieure à p_0 de s'arrêter : en effet, ils ne pourront trouver que des valeurs inférieures à leur valeur courante (principe α-β-minimax). Elle note que tous ces processeurs sont disponibles.
   (d) Elle répartit sur les processeurs disponibles les positions de niveau 1 encore non traitées (la $16^{\text{ième}}$, $17^{\text{ième}}$...).
4. Chaque fois qu'une machine lui communiquera une évaluation partielle inférieure à l'évaluation p_0 elle lui communique l'ordre de s'arrêter et lui donne à évaluer une des positions encore disponibles.
5. Chaque fois qu'une machine lui envoie une évaluation définitive de niveau 1, et que cette évaluation est supérieure à p_0, la machine maître répète les opérations ((a)... (d)).
6. La machine maître poursuit cet algorithme jusqu'à ce que tous les processeurs esclaves soient arrêtés et que toutes les positions de niveau 1 aient été examinées.

Nous avons ainsi décrit une implantation parallèle de l'algorithme α-β[22].

---

22 Il existe d'autres implantations parallèles de l'algorithme α-β beaucoup plus efficaces. Le problème principal de cette implantation est que dans une résolution par un mécanisme α-β, la réponse au premier coup prend généralement 50% du temps de résolution. Donc répartir également chacun des coups du premier niveau est clairement une erreur. L'algorithme PVSA (Principal Variation Splitting Algorithm) améliore le système en parallélisant les réponses pour le premier coup à chaque niveau de l'arbre. On peut se reporter à (AKL *et al.* 1982; Schaeffer 1989a; Marsland and Schaeffer 1990; Kuszmaul 1995) pour des méthodes encore plus efficaces.

15.5.6 Pour conclure ?

La recherche dans le domaine des algorithmes de jeu est d'une richesse et d'une vigueur assez extraordinaire et nous n'avons fait qu'effleurer le domaine. Nous n'avons en particulier pas parlé des alternatives au paradigme minimax, comme les algorithmes de type B^* ou BPIP (Best Play for Imperfect Player). On peut se reporter avec profit à l'excellente thèse de Jean-Christophe Weill, qui a de plus le grand mérite d'être en Français (Weill 1995).

15.6 Othello

Nous nous proposons de développer en détail les différentes phases de la programmation du jeu d'Othello. La programmation d'Othello[23] doit être divisée en trois phases : l'ouverture, le milieu de partie et la fin de partie. Nous allons nous intéresser successivement à ces trois différents types de jeu.

15.6.1 Les ouvertures

Il existe dans le jeu d'Othello un certain nombre d'ouvertures répertoriées. Il s'agit de séquences connues que l'on doit systématiquement utiliser en début de partie, sous peine de se retrouver en position inférieure. Le jeu d'Othello étant beaucoup plus jeune que le jeu d'échecs, la théorie des ouvertures est beaucoup moins développée. On ne s'intéresse à l'heure actuelle qu'à des séquences allant jusqu'à six coups maximum. D'autre part, les séquences recommandées ne sont souvent choisies que pour des raisons statistiques (on sait que, dans un championnat du monde donné, les meilleurs joueurs ont eu leurs meilleurs résultats avec une certaine séquence : c'est donc qu'il s'agit d'une bonne ouverture).

Les ouvertures ne sont pas stockées sous forme de séquence de coups, mais sous forme de positions indexées. Cela signifie que l'ordinateur ne stocke pas des séquences de positions, chaque séquence correspondant à une ouverture, mais stocke séparément chacune des positions avec le coup à jouer si cette position se rencontre à un instant donné. Pourquoi cette méthode ? Simplement pour éviter que l'ordinateur ne se perde dans une interversion de coups. Afin de retrouver rapidement une position dans l'ensemble des positions stockées, on les indexe en général par un certain nombre de clés (nombre de pions de chaque couleur...).

Cette technique a été améliorée considérablement par Michael Buro pour son programme LOGISTELLO, qui est actuellement le meilleur programme mondial. Buro utilise des techniques tout à fait originales pour améliorer en permanence la bibliothèque de son programme. On pourra se reporter avec profit à (Buro 1997) pour plus de détails.

---

23 On peut se demander pourquoi présenter Othello. Plusieurs raisons à cela : c'est un des jeux où les ordinateurs ont obtenu les performances les plus probantes ; un des auteurs de cet ouvrage est également l'auteur du programme d'Othello présenté dans ces quelques lignes, (programme qui battit il y a quelques années un bon joueur français et reste cependant assez loin des tous meilleurs programmes), et connaît donc assez bien les techniques utilisées ; enfin, c'est un jeu relativement connu et dont les règles se comprennent rapidement.

15.6.2 Les finales

Le jeu d'Othello est intéressant dans la mesure où il s'agit d'un jeu à information totale à *nombre de coups précisément connu*. On sait en effet que la partie se terminera au plus tard (et en général exactement) au $60^{\text{ème}}$ demi-coup[24]. En fonction de la puissance de calcul disponible, il est possible de lancer une recherche exhaustive un certain nombre de demi-coups avant la fin (le $60^{\text{ième}}$ coup). Dans ce cas, la fonction d'évaluation est extrêmement simple : il suffit de faire la différence entre son propre nombre de pions et le nombre de pions de l'adversaire. De plus, il s'agit d'une heuristique qui évalue parfaitement le caractère perdant ou gagnant de la position (évidemment !). L'algorithme de recherche est relativement original, puisque l'on considère que le plus efficace dans cet exercice est le $NegaC^*$, un algorithme proche de MTD(f) mais pour lequel on recherche l'encadrement de la valeur de la position par bisection successive de l'intervalle (Weill 1995).

Cette caractéristique du jeu d'Othello explique en partie la force des programmes d'ordinateur : un humain est totalement incapable d'exécuter un calcul total sur une semblable profondeur sans commettre d'erreurs. Ainsi, un ordinateur jouant à Othello peut se retrouver dans une situation difficile une vingtaine de coups avant la fin de la partie et retourner (au propre et au figuré) complètement la position en sa faveur simplement par la force brute[25].

15.6.3 Le milieu de partie

Le milieu de partie est le royaume de l'algorithme α-β et de la fonction d'évaluation. Nous allons examiner en détail la construction d'une fonction d'évaluation trés simple pour Othello. Cette fonction sera une fonction statique dans la mesure où le programme n'est pas capable d'apprentissage. Nous allons nous apercevoir que la construction de la fonction n'est pas chose complètement triviale, même si le jeu est simple et le but raisonnable (il ne s'agit pas de construire un programme champion du monde).

Tout d'abord, on tente d'attribuer à chaque case du jeu une valeur tactique. Cette valeur tactique doit représenter l'intérêt qu'a l'ordinateur à occuper une case ou, au contraire, à la laisser à son adversaire. Une fonction d'évaluation élémentaire consisterait donc à additionner les valeurs des cases que l'on possède et à soustraire la valeur des cases que possède l'adversaire. Une possibilité de valuation des cases est donnée dans la figure 15.8.

On peut rapidement justifier ce tableau en disant que la possession d'un coin est capitale, alors que les cases qui entourent le coin sont à éviter car elles donnent à l'adversaire accès au coin. La possession des cases centrales est importante, car elle augmente généralement les possibilités de jeu. Les bords sont également des points d'appui solides.

Une fois cette évaluation effectuée, il faut la corriger sur-le-champ. En effet, si nous possédons un coin, la valeur des trois cases environnantes doit être considérée comme positive puisqu'elles ne donnent plus accès au coin, qui est déjà occupé. De même, les cases du bord qui sont reliés au coin par une chaîne continue de pions de la même couleur

---

24 À distinguer des échecs où une partie peut durer moins de 10 coups ou plus de 100.

25 On est loin des principes cognitifs fondateurs de l'IA !

500	-150	30	10	10	30	-150	500
-150	-250	0	0	0	0	-250	-150
30	0	1	2	2	1	0	30
10	0	2	16	16	2	0	10
10	0	2	16	16	2	0	10
30	0	1	2	2	1	0	30
-150	-250	0	0	0	0	-250	-150
500	-150	30	10	10	30	-150	500

Figure 15.8 – Exemple de valuation de cases pour Othello

sont beaucoup plus intéressantes car désormais imprenables. Un autre facteur important de la position à Othello est le nombre de cases libres contiguës à ses propres pions. Il faut essayer de le minimiser car plus ce nombre est petit et moins l'adversaire a, en général, de coups disponibles.

15.6.4 Othello et apprentissage

Le programme que nous venons de présenter garantit un niveau de jeu intéressant, mais est cependant très loin des meilleurs programmes d'Othello. Tous les meilleurs programmes utilisent aujourd'hui des techniques d'apprentissage qui peuvent être de plusieurs sortes. Ce type de mécanisme n'est pas nouveau, puisqu'il était déjà utilisé par Arthur Samuel (Samuel 1959) dans son programme de jeu de dames.

La plupart des fonctions d'évaluation sont de la forme $\Sigma c_i t_i$ où les t_i sont les caractéristiques de la position et les c_i les coefficients qui mesurent l'importance à accorder à telle ou telle caractéristique[26]. Au fur et à mesure de l'apprentissage, les coefficients c_i doivent être modifiés de façon à enregistrer l'information acquise pendant les parties.

Comment modifier les coefficients ? Il est clair que l'on doit augmenter la valeur des coefficients qui prédisent correctement l'issue du coup et diminuer la valeur de ceux qui fournissent des informations incorrectes. Le problème dans un programme de jeu est qu'il faut être capable, après la partie, d'analyser les coups qui se sont révélés bons ou mauvais, de façon à modifier la fonction d'évaluation. On peut implanter ce mécanisme en supposant que toute séquence de coups qui a amené à une bonne situation est une bonne séquence et que les coefficients qui ont encouragé cette séquence doivent être renforcés et ceux qui tendaient à l'éviter, diminués (méthode de la carotte et du bâton). Quant à mesurer en quoi une situation est bonne, c'est facile lorsque l'on est proche de la fin de la partie (car on peut aisément faire des analyses exhaustives), mais en milieu de partie, on doit se fier à la fonction d'évaluation elle-même qui fournit, en quelque

26 Nous simplifions ici à l'extrême. En fait, nombre de programmes utilisent des fonctions d'évaluation contenant des termes non linéaires.

sorte, son propre feedback. Ainsi, à chaque coup, Arthur Samuel, dans son programme, comparait la valeur courante de la fonction d'évaluation S_c avec la valeur S_p affectée à cette position lors du calcul au coup précédent (soit deux demi-coups, et donc deux niveaux auparavant). Il en tirait alors les conséquences suivantes :

- Si $S_p > S_c$, il estimait que les termes qui apportaient une contribution négative dans S avaient des coefficients c_i trop faibles et ceux qui apportaient une contribution positive des coefficients trop élevés.
- Si $S_p < S_c$, il en tirait des conclusions directement opposées.

Il ne reste alors plus qu'à modifier les coefficients d'évaluation[27].

Une autre technique classique consiste à faire jouer le programme contre une copie de lui-même avec certains coefficients de la fonction d'évaluation modifiés. Si la nouvelle copie l'emporte, alors on conserve la nouvelle valeur des coefficients, sinon on conserve l'ancienne et on essaie une autre modification.

L'*apprentissage de base*, consiste quant à lui à mémoriser, chaque fois que l'on effectue l'évaluation d'une position, la position ainsi que la valeur associée. Si par la suite, dans une recherche, cette même position est rencontrée, on ne tente pas de l'analyser avec la fonction d'évaluation mais on récupère la valeur stockée en mémoire, qui est évidemment plus précise puisqu'elle a elle-même été évaluée par une recherche α-β profonde. L'inconvénient principal de cette méthode est qu'elle réclame de très importantes capacités de stockage.

La plupart des programmes d'Othello utilisent quand à eux un apprentissage sur des formes standards. Le principe en est simple : on estime (en simplifiant) que la valeur d'une position d'Othello ne dépend que de la structure des 4 bords, des 4 coins 3×3 et du nombre de libertés des pions. On peut alors décider, par exemple, de stocker systématiquement tous les résultats de toutes les parties jouées, et en attribuer le mérite aux structures rencontrées sur les bords et dans les coins au cours de ces parties.

Un programme utilisant correctement de tels concepts devraient figurer dans les dix ou vingt meilleurs programmes mondiaux. Notons pour conclure que le problème de la suprématie homme/machine pour Othello a été réglé de façon à peu près définitive en 1997 où un match opposant LOGISTELLO au champion du monde humain Takeshi Murakami s'est terminé par le score sans appel de 6 à 0 pour le programme.

15.7 Les échecs

Les échecs sont considérés comme le jeu des Rois et surtout le Roi des jeux. Dès les débuts de l'IA, c'est sur les échecs que s'est porté l'effort le plus important et ce sont les échecs qui ont donné lieu à des luttes épiques que se sont livrés les meilleures équipes universitaires comme (HITECH mené par Hans Berliner[28], ou DEEP THOUGHT, mené

---

27 Il peut se produire un problème si la position à laquelle nous nous trouvons n'a pas été étudiée au coup précédent parce que l'algorithme α-β l'a élaguée. Normalement, ce type de situation ne se produit que si l'adversaire fait une erreur, et peut donc être négligé.

28 Hans Berliner présente la caractéristique intéressante d'être un très fort joueur d'échecs, puisqu'il fut champion du monde d'échecs par correspondance. Il s'intéresse à la programmation du jeu d'échec depuis de nombreuses années et passa son PhD en 1974 sur ce sujet. Pour une description de HITECH, voir (Berliner 1990).

par Feng-Hsiung Hsu[29]) devenus depuis DEEP BLUE avec le succès que l'on sait. Il faut sans doute craindre que la victoire de DEEP BLUE face à Gary Kasparov en mai 1997 n'ait des repercussions que l'on pourrait qualifier de néfastes : le plus grand défi que l'on s'était lancé (la victoire d'un programme sur le champion du monde humain dans le cadre de partie à cadence de tournoi et sur six parties) a été réalisé et il est probable que les efforts se porteront davantage dans d'autres directions. D'autre part, même si la lutte entre les plus importantes firmes commerciales restent réels, le niveau des programmes disponibles pour micro ordinateurs est maintenant tellement élevé qu'ils battent systématiquement 99% des joueurs. On accordera donc plus d'importance à la qualité de l'interface avec l'utilisateur qu'à la force pure du programme. Mais les grandes avancées liées aux échecs demeureront.

Les pères fondateurs de l'IA, Simon et Newell, menèrent, ainsi que le psychologue hollandais de Groot et le psychologue et Grand Maître soviétique Nicolaï Kroguious une analyse de la méthode de réflexion des grands champions d'échecs. Nous allons rapidement parler des résultats de ces études avant d'aborder les techniques de programmation proprement dites. Deux raisons à cela : tout d'abord les résultats mis en évidence par de Groot (Groot 1965) et Kroguious (Kroguious 1986) sont très intéressants car ils sont applicables à la plupart des raisonnements humains dans tous les domaines ; enfin, ces résultats n'ont en rien influencé les programmes d'échecs (du moins les meilleurs) qui utilisent des méthodes de raisonnement totalement différentes aujourd'hui, preuve qu'il est bien difficile d'appliquer des techniques humaines pour faire résoudre des problèmes à des machines.

15.7.1 Les études des psychologues

Nous allons ici évoquer les travaux effectués par le psychologue hollandais de Groot[30] et ceux du psychologue et GMI soviétique Kroguious. de Groot (Groot 1965) put travailler avec un échantillon tout à fait remarquable de six GMI (dont un champion du monde, Max Euwe[31]), quatre MI, deux champions des Pays-Bas et dix experts. De par l'exceptionnelle qualité de ces joueurs, l'étude de de Groot est incomparable.

De Groot et ses élèves[32] analysèrent la façon dont les joueurs d'échecs perçoivent une position. Pour ce faire, ils présentèrent à chaque joueur une position sur un échiquier pendant cinq secondes puis enlevèrent les pièces de l'échiquier et leur demandèrent de les replacer. Ils s'intéressèrent alors à deux facteurs : le nombre de pièces replacées correc-

---

29 Pour une description de DEEP THOUGHT, voir (Hsu *et al.* 1990).

30 Cette partie est largement développée, et remarquablement présentée dans (Laurière 1986).

31 Max Euwe, né en 1901 à Amsterdam, fut un très grand théoricien des échecs et remporta le titre mondial en 1935 face à Alexandre Alekhine (profitant, il est vrai, de la mauvaise santé d'Alekhine qui avait, comme souvent, un peu trop forcé sur l'alcool et le tabac). Il le reperdit dès le match revanche, Alekhine, s'étant mieux préparé, l'écrasa très largement. Il tenta de reconquérir son bien en 1948, le titre étant vacant après le décès d'Alekhine, mais ne termina que cinquième du tournoi hexagonal. Il échoua encore plus nettement au tournoi des candidats de 1953 ne terminant qu'avant-dernier. Il se consacra par la suite à la rédaction d'études théoriques, prit part à des études sur le comportement des joueurs d'échecs, et fut élu président de la FIDE en 1970, poste qu'il conserva jusqu'à sa mort. Sa part dans l'étude de de Groot est considérable, d'autant que Max Euwe, ancien professeur de mathématiques, avait un esprit clair et précis dans ses analyses.

32 Comme le fait remarquer (Vicente and Brewer 1993), de Groot n'a réalisé lui-même qu'une seule expérience, celle consistant à faire replacer à tous les cobayes les pièces d'un échiquier dans une configuration valide.

tement et la façon de replacer les pièces. Ils remarquèrent un certain nombre de points intéressants :

- Les GMI parvenaient à replacer l'intégralité des pièces dans près de 40% des cas alors que les experts n'y sont jamais parvenus, tout en obtenant des scores honorables.
- L'ensemble des individus testés qui savaient jouer aux échecs replaçaient les pièces par groupe logique. Ainsi, on replaçait en même temps l'ensemble des pièces qui attaquaient le roque et celles qui le défendaient. Les individus qui ne savaient pas jouer replaçaient les pièces de façon non corrélée et obtenaient des scores désastreux.
- Les grands maîtres avaient en fait réalisé une mémorisation de la position qui était proche d'une analyse. Ils se la rappelaient comme un ensemble de lignes d'attaque et de défense, et non comme un ensemble de pièces.
- Les grands maîtres reconstruisaient la position par référence à des positions de partie déjà connues qui ressemblaient à celle-ci. Ils étaient capables de dégager les grandes lignes du jeu qui avait mené à la position considérée.
- Placés devant des positions aléatoires (non plus tirées de parties réelles, mais composées de pièces posées au hasard sur l'échiquier), tous les individus, joueurs, non joueurs et GMI obtenaient des scores comparables et très mauvais.

D'autres travaux furent menés pour étudier la façon dont les joueurs d'échecs « à l'aveugle[33] » voient mentalement la position. Ainsi un article publié en 1946 par Pina Morini dans la revue italienne *Minerva Medica* estimait qu'il existait différents types de mémorisation de la position, visuelle pour certains et linguistique pour d'autres (qui relisent mentalement la liste des coups joués). En revanche, l'analyse d'une position se fait toujours par bloc logique et par analogie avec des positions connues. Kroguious fait la même remarque en remarquant les effets nocifs du découpage de la position en bloc de pièces. Il signale par exemple le cas suivant.

> « Dans la partie (figure 15.9) Romanovsky-Kasparian (Léningrad 1938), les blancs sont en mauvaise posture. Kasparian, avec les noirs, voulut forcer le mat en créant un réseau à l'aide du Cavalier et de la Dame, et en oublia le reste de l'échiquier. Il annonça mat en trois coups : 52... ♕e1+ ; 53 ♔h2, ♖xh3+ ; 54 ♗xh3, ♘f3 ? ? ?. Romanovsky très gêné, lui expliqua que le Cavalier noir était cloué par la Dame blanche : « Tout d'abord il ne le crut pas, ce n'est qu'en lui montrant du doigt la diagonale a1-h8 qu'il se rendit compte de son erreur. »
>
> Kasparian, tout entier à sa bataille, se désintéressa de certaines pièces, et finit même par violer les règles du jeu. »

Ainsi, il est clair que la mémorisation de la position est fortement liée à la connaissance du sujet, et qu'elle est faite par des schémas de type connexionniste, qui n'ont pas de lien direct avec les techniques de mémorisation des ordinateurs.

De Groot s'intéressa ensuite à la façon de raisonner de chacun de ses cobayes. Il leur présenta des positions et leur demanda quel coup ils joueraient et pourquoi. Il en dégagea les enseignements suivants :

---

33 Jouer à l'aveugle consiste à jouer aux échecs sans voir l'échiquier. Certains joueurs parviennent même à jouer plusieurs parties simultanément sans voir. Ainsi, le grand maître polonais Najdorf joua, en 1947, 42 parties simultanées à l'aveugle, en gagna 36, fit 4 nulles et connut 2 défaites. Le record a été largement battu depuis. Tous les bons joueurs d'échecs ont la capacité de jouer à l'aveugle.

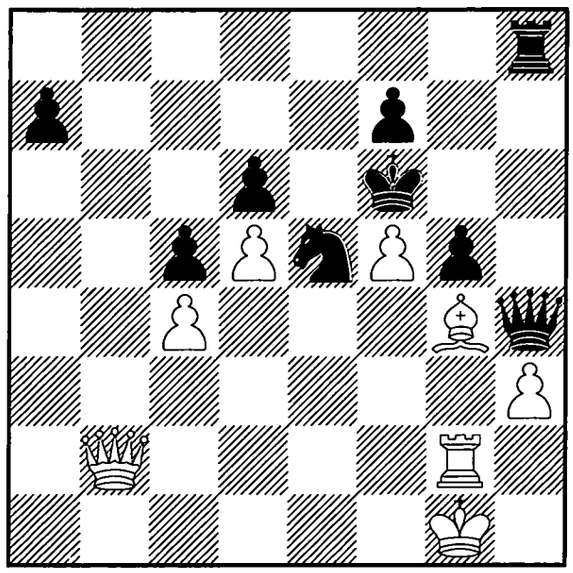


Figure 15.9 – Isolation de sous-blocs dans une position

- La plupart des positions sont tout d'abord « pensées » par analogie avec des positions déjà rencontrées. Ces analogies permettent d'isoler les coups qui peuvent permettre un gain ou éviter une défaite, toujours en rapport avec des stratégies *globales* à long terme.
- La recherche du joueur est très sélective. L'aptitude principale des grands maîtres est leur capacité à identifier rapidement le, ou les bons coups, à analyser sérieusement, précisément par analogie. Une fois ces coups identifiés, tous les autres coups sont ignorés après une analyse superficielle.
- La profondeur du raisonnement est faible. Aucun grand maître n'examine une position à plus d'une dizaine de demi-coups de profondeur (alors que les ordinateurs les plus puissants peuvent examiner jusqu'à quinze ou vingt demi-coups).
- La fonction d'évaluation utilisée est floue ; elle contient des concepts difficiles à identifier clairement, comme la notion de *mobilité utile*, de *pression d'attaque*, ou de *position active* ... Ces critères remplacent l'analyse en profondeur qu'un joueur humain ne peut effectuer en raison de la limitation de ses capacités mentales. Très souvent l'évaluation est aussi effectuée par référence à des schémas connus, faciles à expliciter dans certains cas (position centralisée des Tours, Fous de la « bonne » couleur pour jouer une finale) mais parfois plus indistincts, voire liés au joueur lui-même.
- Le joueur peut effectuer des changements de plan en fonction des analyses qu'il effectue. Ces changements de plan l'amènent alors à reconsidérer les coups à examiner en profondeur.

Les études de Kroguious sont plus orientées vers une analyse « utilitaire » de la psychologie du joueur d'échecs. Capitaine de plusieurs équipes soviétiques[34], il cherche à dégager quels sont les paramètres qui interviennent dans le jugement du joueur d'échecs et renforcent ou dégradent la qualité de son analyse. Kroguious met en évidence l'existence d'images résiduelles, à la fois à long terme et à court terme. L'image résiduelle à long terme permet de reconnaître dans une

34 Il fut le « patron » de l'équipe d'Anatoly Karpov au championnat du monde de Lyon (1990).

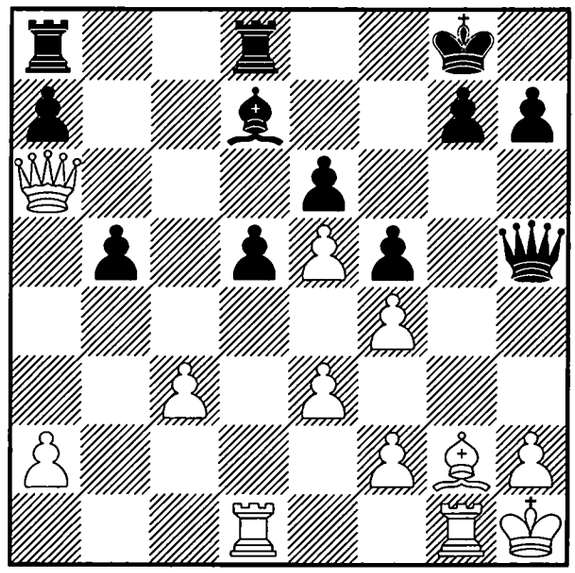


Figure 15.10 – Existence d'images résiduelles à long terme

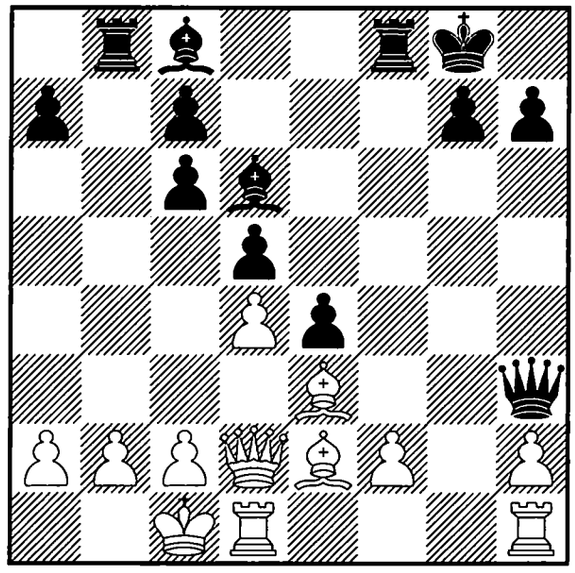


Figure 15.11 – Existence d'images résiduelles

position courante une référence à une position déjà vue, et d'appliquer ainsi immédiatement le même type de plan. Il cite ainsi l'exemple de la figure 15.10.

> « Dans cette position, Bogolioubov avec les blancs trouva instantanément la combinaison 22. ♗xd5,exd5 ; 23. ♖xg7+,♔xg7 ; 24. ♕f6+, ♔g8 ; 25. ♖g1+, ♕g4 ; 26. ♖xg4+, fxg4 ; 25. f5 .
> La clef semble impossible à découvrir si l'on ne se réfère qu'aux principes généraux. Cette combinaison est le produit d'une association d'idées. Inconsciemment, Bogolioubov s'est inspiré d'une partie connue (Bird - Morphy, Londres 1858) (figure 15.11).
> Ici, les noirs jouèrent 17... ♖xf2 ; 18. ♗xf2, ♕a3 avec une attaque gagnante.
> La solution intuitive trouve son origine dans la comparaison entre la position de la partie et une idée familière au joueur. »

Remarquons tout de suite que la similitude que signale Kroguious est loin d'être évidente.

Les études de de Groot et Kroguious sur les joueurs d'échecs sont donc particulièrement intéressantes car elles recouvrent en fait les techniques de raisonnement des humains face à la plupart des problèmes qui leur sont posés : la base de connaissances est fondamentale et l'essentiel de la résolution se fait par référence à des exemples déjà rencontrés[35]. Ceci est rendu possible par la formidable capacité du cerveau humain à stocker et trier les informations, et à retrouver instantanément (ou presque) l'ensemble des informations relatives à un objet donné, pourvu que l'on fournisse un identificateur suffisamment clair de l'objet. Ce qu'il y a parfois de plus remarquable encore, c'est que nombre d'informations peuvent disparaître, mais certaines associations apparemment sans valeur subsister[36]. Il faut d'ailleurs noter la fantastique quantité d'informations que le cerveau emmagasine tout au long d'une existence. Il faut également souligner que la capacité à mémoriser les informations, si elle est importante, n'est pas primordiale. Ce qui est réellement important c'est la capacité que l'on a à récupérer efficacement ces informations quand on est confronté à certains indices. Tous les enseignants savent bien que faire comprendre quelque chose est important, mais qu'il existe certains élèves qui, bien qu'ayant compris, restent incapables d'appliquer la méthode apprise dès que le problème est légèrement modifié dans sa présentation[37]. La technique est mémorisée, mais le cerveau est incapable de reconnaître l'analogie des situations. Il nous est impossible, à l'heure actuelle, de modéliser[38] comment le cerveau détecte les analogies[39]. Ce fut la cause de l'échec du GPS de Newell, Simon et Shaw et reste le problème majeur à résoudre aujourd'hui. Si on savait le faire, on serait non seulement capable de construire des ordinateurs reproduisant le raisonnement humain, mais on serait peut-être aussi capable de rendre les humains plus intelligents[40] en développant les capacités de leur cerveau à raisonner correctement[41].

Après ces digressions revenons à notre sujet principal : la programmation des échecs.

15.7.2 Les ouvertures

La programmation de l'ouverture est certainement la chose la plus simple à réaliser. Les techniques à utiliser sont exactement les mêmes que pour la programmation des ouvertures pour Othello. Cependant, le nombre de positions aux échecs rend le problème un peu plus complexe. Il faut savoir que l'encyclopédie des ouvertures doit dépasser les 2000 pages.

---

35 C'est finalement l'histoire du polytechnicien à qui l'on demande de faire chauffer de l'eau chaude après lui avoir expliqué comment faire chauffer de l'eau froide : il suffit de laisser refroidir l'eau chaude et *l'on est ramené au problème précédent* : la phrase magique du taupin résolvant un problème.

36 C'est ainsi qu'un mot, une odeur, une image peuvent déclencher des sensations de « déjà vus » sans qu'il soit possible d'identifier complètement à quoi cette sensation se rapporte exactement.

37 Un professeur de mathématiques prétendait même présenter à chaque interrogation le même problème sous un aspect différent et avoir à la fin de l'année une fraction d'individus toujours incapable de le résoudre.

38 À l'exception peut-être des réseaux de neurones formels, mais le modèle semble tout de même bien pauvre.

39 Pour exprimer un avis personnel, je pense, comme Herbert Dreyfus, que cela rend, pour l'instant, irréaliste toute tentative de construction de programme reproduisant le fonctionnement d'un raisonnement humain.

40 Ce qui justifie la phrase de Patrick Winston : « Rendre les ordinateurs intelligents peut aider à rendre les hommes plus intelligents. »

41 Je crains fort hélas que les capacités des cerveaux à réaliser des analogies soient plus du domaine de la biologie que du domaine des modèles informatiques ou psychologiques.

Il s'agit plus d'un travail de bénédictin que de la réalisation exaltante d'algorithmes évolués. Notons une fois de plus la nécessité de ne pas stocker les séquences de coups, mais bien les positions en les indexant de façon correcte, afin de ne pas être trompé par une inversion de coups. Certains programmes se font encore duper. On peut aisément le vérifier avec la séquence suivante :

1.	e2–e4	e7–e6
2.	♘g1–f3	c7–c5

Il suffit alors de demander à l'ordinateur le coup qu'il jouerait face à la position résultante. S'il répond immédiatement d4, c'est qu'il a reconnu l'inversion de coup qui a transformé une partie française en une sicilienne. La séquence classique pour atteindre cette position sicilienne est en effet : 1. e4,c5 ; 2.♘f3,e6. Le coup e6 en premier par le joueur noir fait au contraire croire à un ordinateur primaire que l'on va jouer une partie française, auquel cas il attend comme séquence standard : 1. e4,e6 ; 2. ♘f3,d5.

Ne trouvant pas cette séquence standard (c5 au lieu de d5), il est perdu s'il stocke ses ouvertures sous forme de séquences de coups et non sous forme de positions indexées, car il ne reconnaît aucune séquence standard et est incapable de reconnaître une position (sicilienne).

Une technique classique consiste, pour accélérer la recherche, à indexer la position sur le numéro du demi-coup où on peut la rencontrer. On peut, bien entendu, écrire un programme qui, connaissant les séquences d'ouverture, génère la structure de données appropriée pour faire de la reconnaissance de position, la réalisation d'une telle structure par un être humain étant réellement fastidieuse.

En fait, une technique plus efficace encore consiste à réaliser un programme qui utilise conjointement les deux méthodes : mémorisation des séquences d'ouverture pour une plus grande rapidité et mémorisation des positions pour la généralité. Il n'aurait recours à la seconde méthode que lorsque la première ne lui permettrait pas de conclure efficacement.

Hormis ces détails techniques, la programmation des ouvertures ne pose aucun problème difficile.

15.7.3 Les finales

Les finales furent longtemps le talon d'Achille des programmes d'échecs avant de devenir un de leur point relativement fort.

Quel problème avait-on à résoudre ? Les ordinateurs sont efficaces en milieu de partie en raison de leur formidable habileté tactique, liée à leur grande capacité de calcul et à leurs algorithmes de recherche efficaces. Mais dans une finale, la capacité à calculer doit être bien différente. Prenons un exemple trivial. Si nous sommes dans une finale Roi et pion contre Roi et pion, et que la disposition des pièces est la suivante : pion blanc en a2, Roi noir en g4, pion noir en h2, Roi blanc en h1 avec trait au blanc (figure 15.12).

La partie peut être considérée comme terminée. En effet, le Roi noir ne peut empêcher le pion blanc d'aller à Dame en a8 et le pion noir en h2 ne peut aller à Dame, bloqué qu'il est par le Roi blanc en h1. C'est l'analyse que réalise immédiatement tout joueur, même débutant. Le bon coup est évidemment a4. Mais un ordinateur raisonnant avec un algorithme α-β à profondeur fixe va considérer les choses différemment. Tout

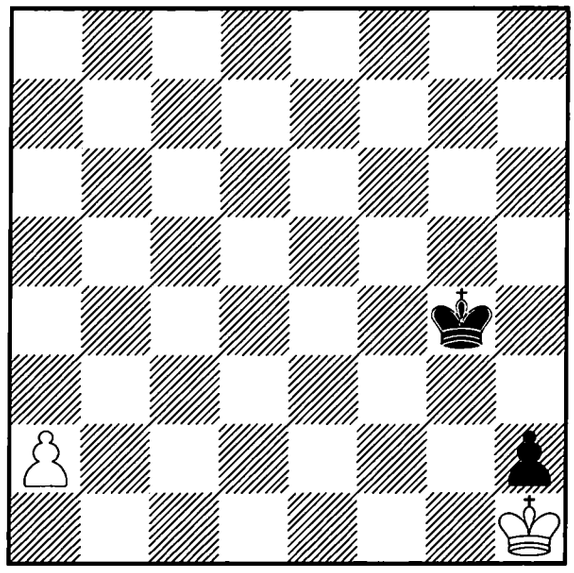


Figure 15.12 – Effet horizon dans une finale

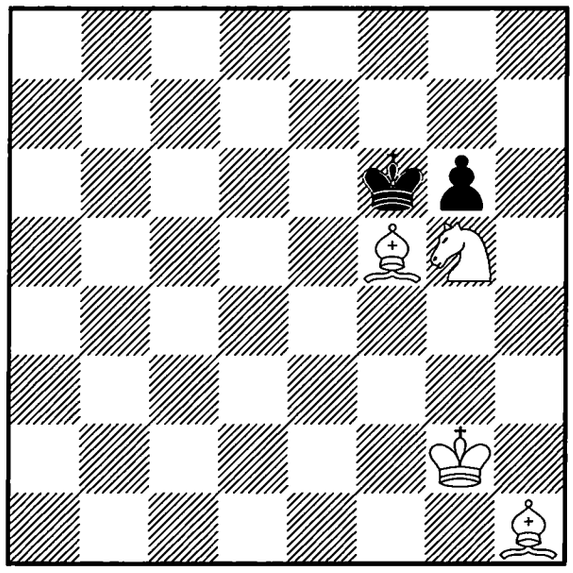


Figure 15.13 – Reconnaissance d'une finale Fou-Cavalier

d'abord, il voit qu'il peut prendre le pion h2 sur le champ, et qu'en revanche s'il ne le prend pas, il ne pourra pas le prendre au coup suivant si les noirs jouent Th3, il y a donc manque à gagner. D'autre part, il lui faudrait analyser la position à 10 demi-coups de profondeur pour s'apercevoir que le pion blanc ne peut aller à Dame que s'il est avancé sur le champ. Cette profondeur étant déraisonnable pour beaucoup d'ordinateurs, il sera aveuglé par sa gloutonnerie et jouera |Rxh2|, la partie se terminant alors par une nulle (le Roi noir capturera le pion blanc sans difficulté au bout de 10 demi-coups). Cet exemple nous montre la nécessité d'*inclure dans le programme des règles particulières permettant de déterminer aisément la valeur d'un pion dans une finale sans être contraint à des analyses en profondeur trop importantes.*

Un autre exemple typique de problème de finales est le suivant. On considère la position : Roi noir en f6, pion noir en g6, Roi blanc en g2, Fou blanc en f5 Cavalier blanc en g5 et Fou blanc en h1 (figure 15.13).

Le programme ne peut défendre simultanément son Fou en f5 et son Cavalier en g5. Il jouera donc ♗xg6, gagnant un pion avant de perdre le Fou. Il se retrouve alors

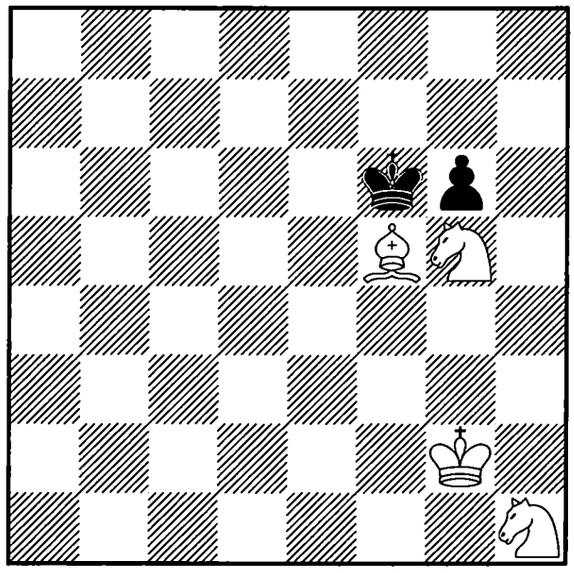


Figure 15.14 – Reconnaissance de la finale Cavalier-Cavalier

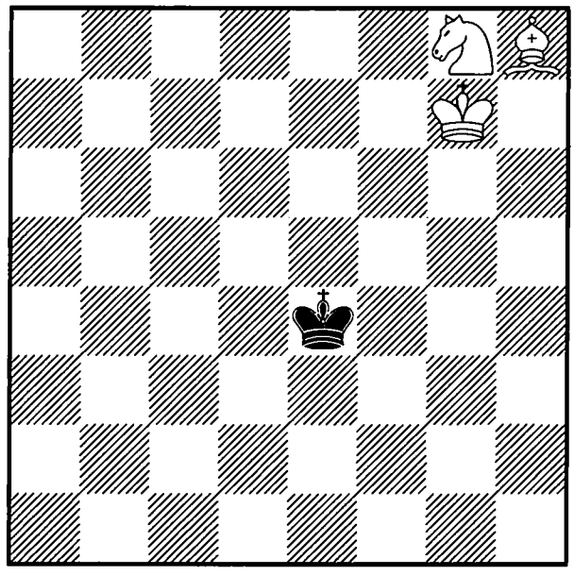


Figure 15.15 – Établir un plan

à jouer une finale Roi, Fou, Cavalier contre Roi qui est gagnante. Jusque-là pas d'erreur. Remplaçons maintenant le Fou blanc en h1 par un Cavalier blanc(figure 15.14).

Le problème est apparemment inchangé, la meilleure solution semble encore de jouer |Fxg6| pour gagner un pion avant de perdre une pièce. C'est d'ailleurs ce que font tous les programmes de l'ancienne génération et nombre de joueurs d'échecs novices. Il s'agit pourtant d'une erreur grave. En effet, la finale Roi, Cavalier, Cavalier contre Roi est toujours nulle. Le coup correct ici est |Fd3|, pour sauver le Fou. Après |Rxg5|, les blancs élimineront sans difficulté le pion noir restant avec leur coalition Roi, Fou, Cavalier puis materont le Roi noir. Mais là encore, il faut savoir sacrifier le court terme pour le long terme et surtout il faut savoir *reconnaître certains types de positions particulières.*

Signalons un dernier point capital dans beaucoup de finales : *la nécessité d'établir un plan.* Considérons la position suivante : Roi noir en e4, Roi blanc en g7, Fou blanc en h8, Cavalier blanc en g8 (figure 15.15).

Il s'agit d'une typique finale Roi, Fou, Cavalier contre Roi. Cette finale est gagnante. Encore faut-il mater en moins de 50 coups[42]. Nombre de joueurs de club ont échoué dans une semblable finale et tous les programmes de l'ancienne génération échouaient également. Le mat ne peut être atteint que par un jeu précis et un plan parfaitement établi. Un problème encore plus difficile est posé par la finale Roi, Cavalier, Cavalier contre Roi et Pion qui peut être gagné dans certains cas[43]. Mais la réalisation du mat en 50 coups est excessivement difficile sans plan précis.

Nous avons donc vu la nécessité d'introduire trois éléments fondamentalement nouveaux dans les programmes de finale :

- des règles particulières permettant de calculer instantanément la valeur de certaines pièces (pion passé, règle du carré, notion de « bon Fou »...) ;
- savoir reconnaître des types de positions classiques ;
- savoir appliquer des plans précis adaptés à des cas précis.

Les progrès réalisés ces cinq dernières années par les programmes en résolution de finales sont extrêmement spectaculaires[44]. Tous les exemples que nous avons présentés ci-dessus sont maintenant résolus par des programmes que l'on peut trouver dans le domaine public sur internet. Deux facteurs ont largement contribué à la progression des ordinateurs dans les finales : les tables de transposition (que nous avons déjà présentées), et l'analyse rétrograde que nous allons maintenant détailler.

Il s'agit d'une méthode permettant de constituer des bases de données indiquant à l'ordinateur quel est le coup à jouer dans une position donnée pour arriver au gain de la façon la plus rapide. La constitution de semblables bases s'est beaucoup développée dans les dix dernières années. Ainsi, il existe une base de données permettant de jouer parfaitement des finales comme (Roi + Cavalier + Fou) contre (Roi + Cavalier), une finale que même un champion du monde aurait des difficultés à gagner[45].

Pour construire de semblables bases de données, les ordinateurs utilisent une méthode appelée *analyse rétrograde*. Prenons comme exemple la finale (Roi + Dame) contre (Roi). Il existe, si on néglige les symétries, $64 \times 64 \times 64 = 262144$ positions de départ. Certaines de ces positions sont impossibles (les deux Rois adjacents par exemple) : elles sont éliminées. Dans les positions restantes, on recherche celles où les noirs sont mats : ce sont les positions où le Roi noir est déjà en échec et où aucun des coups dont il dispose ne permet de le soustraire à cet échec. À partir de là, on détermine aisément les positions de mat en un coup. Ce sont les positions qui permettent au blanc d'atteindre en un coup une position de mat. On peut ainsi déterminer récursivement les positions de mat en $n + 1$ coups à partir des positions de mat en n coups.

---

42 Rappelons qu'il existe aux échecs une règle qui stipule qu'une partie est nulle s'il se produit une séquence de cinquante coups sans prise de pièce ou de pion.

43 C'est une des bizarreries des échecs. La finale Roi, Cavalier, Cavalier contre Roi est nulle, mais la finale Roi, Cavalier, Cavalier contre Roi et Pion peut être gagné, le pion noir étant utilisé par les blancs pour bloquer une case de fuite du Roi noir.

44 Une des critiques que l'on peut adresser à la présentation des techniques de jeu dans (Laurière 1986) est qu'elle date sérieusement par rapport aux résultats actuels.

45 L'ordinateur a d'ailleurs prouvé que les règles actuelles du jeu d'échecs étaient, dans un certain sens, « incorrectes ». En effet, la règle stipule qu'une partie est déclarée nulle si aucune prise de pièces ou aucune promotion n'a eu lieu en cinquante coups. Or, il existe des cas où la finale (Roi + Cavalier + Fou) contre (Roi + Cavalier) ne peut être gagnée qu'en soixante-dix-sept coups.

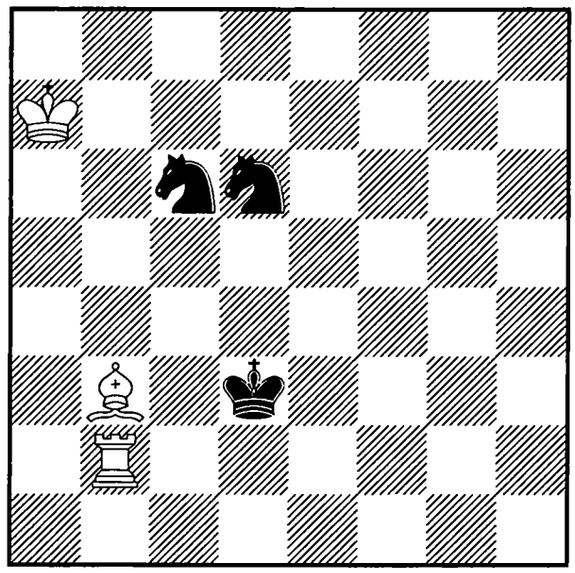


Figure 15.16 – Position initiale de la plus longue finale connue

La constitution de bases de données de ce type est une technique bien particulière, relativement éloignée de la programmation classique des jeux, mais fort intéressante. Les problèmes à 5 pièces (sans pion) ont été exhaustivement résolus[46], et l'on travaille activement sur les problèmes à 6 pièces. En Octobre 1991, Larry Stiller (avec l'aide de Noam Elkies et Burton Wendroff) a construit la plus longue finale gagnante connue à ce jour. La position initiale est représentée sur la figure 15.16. La résolution complète demande 223 coups. Il est clair que ce type de finale est inaccessible à l'être humain. La question est de savoir si les règles doivent être modifiées pour en tenir compte.

15.7.4 Le milieu de partie

Nous allons retrouver les deux grands protagonistes du milieu de partie : la fonction d'évaluation et l'algorithme α-β. Mais nous introduirons également quelques techniques spécifiques ou, en tout cas, surtout utilisées pour les échecs.

La fonction d'évaluation aux échecs est extrêmement complexe. La base d'une fonction d'évaluation est la valeur accordée aux pièces. On peut dire que les valeurs que la plupart des programmes accordent aux pièces sont les suivantes :

Dame	:	9
Tour	:	5
Fou	:	3
Cavalier	:	3
Pion	:	1

Il reste maintenant à raffiner la fonction d'évaluation. Tout un ensemble de paramètres interviennent. En voici quelques uns :

---

46 Pour être tout à fait précis, Larry Stiller calcula exhaustivement les 80 positions les plus intéressantes en 227 minutes sur une CONNECTION-MACHINE (TMC 1991) à 32000 processeurs (une demi-CONNECTION-MACHINE en fait).

- mettre son Roi en sécurité (Roquer, éviter d'affaiblir le roque par des avancées de pions) ;
- conserver la paire de Fous ;
- dominer le centre ;
- garder un maximum de mobilité ;
- centraliser les Cavaliers ;
- placer les Fous sur des diagonales ouvertes ;
- placer les Tours sur des colonnes ouvertes ;
- doubler les Tours sur les colonnes ouvertes ;
- conserver une bonne structure de pions (pas de pions doublés, conserver des pions liés, éviter les pions arriérés) ;
- obtenir des pions passés soutenus (capital en vue de la finale) ;
- conserver le Fou de la « bonne » couleur (un Fou est de la bonne couleur lorsqu'il peut attaquer les cases où sont bloqués les pions de l'adversaire. C'est un concept fondamental pour avoir un avantage en finale.) ;
- favoriser les échanges si le passage en finale est favorable.

Ce n'est qu'un petit aperçu de ce que doit être une bonne fonction d'évaluation aux échecs.

Notons cependant un point capital : *la notion de plan stratégique d'ensemble est absente.* On peut, certes, introduire dans la fonction d'évaluation quelques éléments pseudo-stratégiques, comme l'attaque sur le roque adverse « à la baïonnette[47] », mais cela reste très limité. Il s'agit bien là du principal défaut des programmes d'ordinateurs. Cela est particulièrement sensible lorsqu'ils passent de la séquence d'ouverture qu'ils jouent « par cœur » au milieu de partie. En effet, à chaque ouverture est associée une idée stratégique et la suite de la partie doit développer cette idée. Ainsi une sicilienne doit être jouée de façon agressive par les blancs qui doivent attaquer rapidement le petit roque noir, alors que la chance des noirs se situe en général à l'aile dame. Or un programme est incapable de saisir ces subtilités stratégiques et appliquera la même fonction d'évaluation quelle qu'ait été l'ouverture. Tous les programmes d'ordinateur, même les meilleurs, manquent de « liant » dans leur jeu, ce qui rend leur style facilement reconnaissable et les laisse encore vulnérables. On ne connaît actuellement aucune méthode pour remédier à cet état. Simplement, la force tactique des programmes leur permet de compenser leur faiblesse stratégique.

La technique de recherche dans l'arbre de jeu est un algorithme α-β tel que nous l'avons décrit au paragraphe précédent. La technique de contrôle de temps de jeu et de reclassement des coups avant d'aborder des recherches plus profondes est également identique. Il faut cependant introduire certaines techniques indispensables aux échecs[48] :

L'élagage de futilité : Nous savons que l'algorithme α-β élague une branche au niveau n dès que l'évaluation partielle de cette branche est inférieure (ou supérieure si le niveau est pair) à l'évaluation définitive du niveau $n-1$. Supposons par exemple que l'évaluation du niveau $n-1$ soit 3 et qu'une évaluation partielle au niveau n soit 3.01. Certes, l'évaluation n'est pas inférieure, mais il est clair qu'elle ne sera

47 Technique consistant à attaquer le roque adversaire avec les pions.

48 Ces techniques peuvent s'appliquer, avec plus ou moins de bonheur ou d'utilité, à d'autres jeux.

pas non plus franchement meilleure. On peut donc élaguer cette branche. Cette technique est appelée *élagage de futilité.*

Le coup meurtrier : Supposons que lors de la recherche α-β, l'algorithme note que le coup de premier niveau p_1 est détruit par la riposte de d_1 (c'est-à-dire que si l'adversaire répond d_1 après p_1 le score de la position s'effondre). Cela signifie que d_1 est un *coup meurtrier.* Lors de l'examen du coup de premier niveau suivant p_2, la première riposte que l'algorithme α-β devra examiner sera précisément (si cela est possible) d_1 car si le coup d_1 a été meurtrier pour p_1, il a de fortes chances de l'être également pour p_2. De tels coups sont liés, aux échecs, à la notion de menace et la même menace reste souvent valable face à de nombreux coups. En procédant de cette façon, on améliore encore l'efficacité de l'élagage de l'algorithme α-β.

Utilisation du temps de réflexion de l'adversaire : Après avoir joué un coup, il est bon de retenir l'intégralité de la branche. En effet, on peut relancer la recherche α-β *pendant le temps de réflexion de l'adversaire* sur la position telle qu'elle se produirait si l'adversaire joue la riposte que nous estimons la meilleure pour lui. De cette façon, si l'adversaire joue effectivement ce coup, nous aurons déjà effectué une grande partie de l'évaluation de la position.

Retour à l'équilibre : Si l'algorithme α-β travaille à une profondeur fixe, il s'expose à de sérieux ennuis. Supposons en effet que l'algorithme α-β descende à une profondeur 3 et qu'il trouve une séquence du genre : je déplace mon Cavalier (premier niveau), on me le prend (deuxième niveau), je prends la Dame adverse (troisième niveau). Bien qu'il enregistre la perte de son Cavalier, le résultat est globalement positif puisque le programme comptabilise la prise de la Dame adverse. Mais supposons que le quatrième coup (non examiné puisqu'il s'est arrêté à la profondeur 3) est la prise de sa Dame. Le bilan global de l'opération est là franchement négatif. Pour éviter ce type de comportement, tous les algorithmes utilisent une stratégie dite *de retour à l'équilibre.* Une fois atteint leur profondeur maximale d'évaluation, ils poursuivent le calcul de la position terminale en continuant à examiner toutes les positions (et seulement les positions) découlant de prises ou d'échecs, jusqu'à ce qu'il n'y en ait plus. Ils peuvent ainsi éviter ce type de mauvaise évaluation.

Recherche secondaire : Un autre problème qui guette les ordinateurs est *l'effet horizon.* Voici un exemple classique dû à Hans Berliner (cf. figure 15.17).

Le Fou blanc en a4 est perdu. En effet, il ne pourra échapper au pion b5 ou au pion c5 (qui va venir en c4). La séquence qui provoquera la perte du Fou est ♗b3 suivi de c4 par les noirs et le Fou est perdu. Mais un algorithme α-β tente simplement de repousser cette éventualité au-delà de son horizon de recherche (d'où le nom d'*effet horizon*). Il va donc tenter de jouer une séquence de coups à riposte forcée qui repoussera hors de son champ de vision la séquence « déplaisante ». Dans le cas présent, il prévoit par exemple (profondeur 5) : 1. e5,d×e5 (obligatoire car sinon le Cavalier en f6 tombe), 2.♘d5,♘×d5 (on ne peut jouer 2. . . .,b×a4 car alors 3.♘×e7+), 3.♖×d5.

Arrivé à cette position, le programme applique son mécanisme de retour à l'équilibre. Mais il ne voit rien de dangereux car après 3. . . .,b×a4 il voit 4. ♖×d7 et récupère le Fou. En fait, son raisonnement est incorrect car les noirs intercalent le coup intermédiaire ♗e6 et le jeu se poursuit en fait par : 3. . . .,♗e6 ; 4.♗b3,c4 et le Fou est perdu.

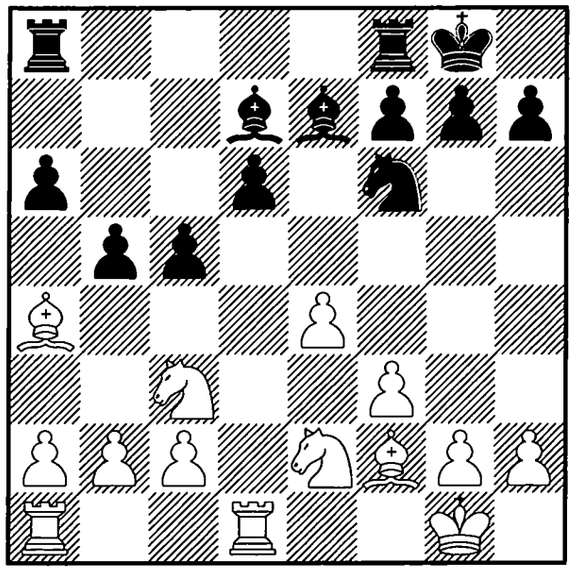


Figure 15.17 – Effet horizon lié à un coup intermédiaire

Le programme, pour trouver la solution correcte, devrait effectuer la recherche à la profondeur 7, ce qui est prohibitif pour un ordinateur moyen. Certains écrivaient, il y a quelques années, que ce type de situation était ingérable par un ordinateur. En fait, la plupart des bons programmes de jeu utilisent aujourd'hui une technique appelée *recherche secondaire* qui leur permet de trouver la solution du problème précédent en quelques secondes. Il s'agit simplement de relancer la recherche α-β à une profondeur supérieure, mais uniquement sur la branche (voire uniquement sur la position terminale de la branche) qui contient le coup sélectionné. Il s'agit, en quelque sorte, d'une vérification de la validité du coup choisi. Comme l'évaluation n'est faite que sur la position terminale sélectionnée, le temps nécessaire est raisonnable. Certes, ce mécanisme ne permet pas de faire disparaître l'effet horizon, mais il en limite considérablement les effets[49].

L'algorithme SEX : L'algorithme SEX (Search EXtension) (Levy *et al.* 1989) a aussi pour but de limiter les problèmes d'effet horizon. Dans un mécanisme de recherche minimax classique, la profondeur de recherche est fixée au départ à n et à chaque descente d'un niveau dans l'arbre, on diminue de 1 la valeur de n. Lorsque n atteint 0, la recherche est terminée.

Avec l'algorithme SEX, on utilise une variable *SX* qui prend une valeur nettement plus grande que la profondeur à laquelle on veut chercher (une valeur typique est dix fois la profondeur, si n vaut 3, *SX* vaut 30), mais à chaque descente d'un niveau dans l'arbre, on diminue *SX* d'une quantité variable en fonction de l'intérêt du coup joué. Ainsi, un coup « moyen » soustrait 10 à *SX*, une prise ou un échec est un coup intéressant, et on diminue peu *SX* (2 ou 3) ; en revanche, une retraite de pièces est un coup peu intéressant et on diminue beaucoup *SX* (jusqu'à 35). La recherche est terminée quand *SX* devient négatif ou nul.

De cette façon, les coups intéressants sont étudiés de façon plus profonde, et les coups « peu intéressants » sont rapidement abandonnés.

---

49 Sur les techniques dites de *Singular Extension*, voir (Anantharaman *et al.* 1990).

L'heuristique du « coup nul » : Appelé *Null Move Heuristic* en anglais, cette technique consiste à supposer que, dans une situation donnée, un joueur exécute deux coups successifs[50] (son adversaire ne joue pas, d'où le nom de « coup nul »). Si la situation ainsi générée n'est pas clairement favorable, alors le premier coup est élagué de l'arbre de recherche. Cette technique permet d'améliorer l'efficacité de l'élagage mais provoque parfois des élagages intempestifs, comme un match du programme Mephisto au tournoi ACM de 1991 le montra...

Signalons, pour conclure, un des inconvénients de la méthode minimax. Si l'ordinateur se trouve dans une situation perdante et qu'il a le choix entre deux coups, dont le second est légèrement plus mauvais que le premier mais tend un piège à l'adversaire, il sera incapable de s'en apercevoir. Le principe minimax choisira obligatoirement le premier, alors qu'il garantit presque assurément la perte de la partie, alors qu'un joueur humain tentera le tout pour le tout et jouera le second coup. La notion de piège est bien difficile à formaliser, cependant Donald Michie proposa, il y a quelques années, une intéressante méthode qu'il est bon de rapporter. Supposons que nous ayons un arbre à deux niveaux que nous parcourons par une méthode minimax. L'ordinateur cherche la valeur S_1 du premier coup de niveau 1. À partir de la position P_1 correspondant à ce coup, son adversaire peut atteindre trois positions P_{11}, P_{12} et P_{13} dont les scores sont respectivement S_{11}, S_{12}, S_{13}. Théoriquement, l'ordinateur devrait remonter comme valeur au niveau 1 le minimum de ces trois valeurs. Michie propose de raffiner la méthode en prenant comme valeur du niveau 1 non pas le minimum, mais la combinaison linéaire :

$$p_{11} \times S_{11} + p_{12} \times S_{12} + p_{13} \times S_{13}$$

p_{11}, p_{12}, p_{13} représentent les probabilités que l'adversaire a de jouer les coups S_{11}, S_{12}, S_{13} d'après l'ordinateur.

Comment évaluer ces probabilités ? Michie a mis au point une méthode pour les échecs, qu'il appelle *modèle de discernement.* Il estime que le discernement d'un joueur face à une position donnée est donné par la formule empirique :

$$d = (C/1000 + 1)3^{1/(n+0.01)}$$

C représente le classement ELO[51] du joueur que l'on affronte et n le nombre de coups dont on a calculé une valuation. Pourquoi cette formule ? Il est clair qu'un joueur joue d'autant mieux qu'il est plus fort. D'autre part, nous savons que plus il y a de coups différents possibles et plus le choix est difficile, surtout s'ils ont des valeurs proches. Michie estime alors que la probabilité qu'un joueur de discernement d face à une position joue un coup menant à un score estimé v sera :

$$p = Ad^v$$

---

50 Pour une description précise, voir (Goetsch and Campbell 1990).

51 Le classement ELO d'un joueur est une procédure internationale permettant d'attribuer une force à un joueur d'échecs. La méthode de calcul de ce nombre de points est complexe. Disons que l'on gagne des points chaque fois que l'on bat ou que l'on annule contre un joueur plus fort que soi, et que l'on en perd dans le cas contraire (présentation fort sommaire et quelque peu inexacte). Pour donner une idée, le champion du monde actuel a un ELO de l'ordre de 2800, un GMI tourne autour de 2500, un MI autour de 2200, la moyenne de l'ensemble des joueurs d'échecs est d'environ 800. Les meilleurs programmes commerciaux actuels ont une force de l'ordre de 2500/2600 points.

où A est un coefficient bien choisi pour normer la probabilité. Il serait intéressant de tester la méthode de Michie, mais aucun programme ne semble l'utiliser aujourd'hui.

Enfin, et d'après[52] Feng-Hsiung Hsu (responsable du projet DEEP THOUGHT), les programmes d'échecs à l'heure actuelle n'utilisent pas de technique d'apprentissage[53] :

> « Une grave erreur doit être évitée : DEEP THOUGHT n'apprend pas. La fonction d'évaluation que nous utilisons est linéaire par rapport à presque tous ses termes (de 100 à 150). Nous ne faisons qu'une optimisation du modèle de mérite dans un espace multi-dimensionnel, en utilisant un mécanisme de régression linéaire. Ce processus produit des résultats intéressants, même s'il n'est pas clair qu'il produise toujours les effets désirés. »

Pour plus d'informations sur les programmes qui jouent aux échecs, on ne peut que conseiller (Levy and Newborn 1991; Marsland and Schaeffer 1990).

15.8 Le bridge

Notre exposé sur la programmation du bridge sera moins détaillé et moins complet que celui consacré aux échecs. La raison en est simple : peu de gens se sont intéressés à la programmation du bridge. On peut penser que la raison de ce désintérêt est l'aspect « chance » qui existe au bridge. Les gens pensent que si un programme d'échecs l'emporte, il est meilleur, alors qu'un programme de bridge peut simplement « avoir de la chance ». Il y a également le facteur « convivialité » du bridge : on ne joue aux échecs qu'avec un partenaire, alors que le bridge se joue à quatre et on joue souvent au bridge plus pour être entre amis que pour améliorer son niveau de jeu. Enfin, une raison technique est que la réalisation d'un programme, même élémentaire, est beaucoup plus difficile qu'aux échecs : il faut en effet écrire un programme d'annonces correct, ce qui n'est déjà pas simple, puis introduire, au lieu des mécanismes de recherche classiques, des analyses probabilistes portant à la fois sur les informations fournies par les annonces et sur les informations fournies par le jeu de la carte. Il faut également utiliser des notions de planification pour choisir, avant de commencer le jeu de la carte (du moins pour le camp déclarant), la façon dont on estime pouvoir réaliser son contrat. En fait, tous les jeux à information incomplète sont difficiles à programmer, car il est impossible d'utiliser directement des méthodes de recherche classiques, en raison du très important nombre de distributions possibles de mains.

15.8.1 Les annonces

Le premier problème qui se pose au bridge est de construire un programme d'annonces efficace. Les annonces au bridge sont quelque chose de complexe et de très codifié. On pourrait penser qu'il est raisonnable de créer un système capable de contenir l'ensemble des situations possibles. C'est en fait irréalisable. Il faut donc construire un

---

52 Correspondance privée.

53 Sans qu'il soit toujours simple de préciser ce qui est ou n'est pas de l'apprentissage. Il est souvent très tentant, et dans tous les domaines, d'utiliser ce genre de mot, à la sémantique « attirante » et mal définie, pour séduire les financeurs potentiels. Ceux-ci devraient, comme dans les restaurants, se méfier des menus aux noms pompeux et aller faire un tour en cuisine...

système à mi-chemin entre le « par cœur » et des mécanismes heuristiques plus généraux. Un exemple typique de la difficulté de la programmation des enchères au bridge est le suivant : Nord annonce 1♠. Cela signifie qu'il a entre 13 et 20 points distribution-honneurs[54] et au moins cinq Piques. Il attend alors la réponse de son partenaire Sud. Les réponses de Sud sont également codifiées de façon très précise comme l'est la première enchère. Mais supposons qu'Est intervienne entre Nord et Sud. Voilà Sud dans une situation bien difficile. Il devra moduler sa réponse en fonction de l'intervention d'Est. Si Est a simplement dit 1SA[55], il pourra reparler au niveau de 2 et ne sera privé que de l'enchère de 1SA. En revanche, si Est a fait une enchère de barrage au niveau de 4, le problème est bien différent.

Le problème des enchères est encore compliqué par le fait qu'il existe plusieurs systèmes d'enchères différents, ainsi que des conventions spéciales que le programme doit également connaître.

Enfin, un bon programme doit également savoir utiliser les informations que donnent les enchères pour mettre à jour les probabilités de présence des cartes chez les autres joueurs. Ainsi, supposons qu'un des adversaires ouvre de 1♠. Nous savons qu'il a entre 12 et 20 points d'honneur[56] et au moins cinq Piques. Nous pouvons donc modifier les probabilités de présence des cartes dans sa main. Jusque-là, pour toute carte que nous ne possédions pas, nous pouvions supposer que chacun des autres joueurs avait une probabilité de 1/3 de la posséder. Si je possède moi-même 10 points d'honneur, il en reste 30 dans le jeu. Le joueur qui a ouvert doit avoir en moyenne 13 points d'honneur[57], donc je dois désormais donner à chacun des honneurs qu'il peut avoir dans sa main, une probabilité de $13/30 = 0.43$. Nous devons, en conséquence, modifier la probabilité de présence de ces mêmes cartes dans toutes les autres mains. Le processus doit se poursuivre tout au long des enchères. À la fin, un bon programme doit avoir accumulé suffisamment d'informations pour pouvoir planifier correctement son jeu.

15.8.2 Le jeu de la carte

Le jeu de la carte est excessivement difficile à programmer. Il faut en effet planifier le jeu avant l'entame, modifier les plans en cours de partie si les défendants ne suivent pas la ligne prévue ou que les cartes ont une distribution défavorable, déclencher une recherche exhaustive quand la position de l'ensemble des cartes est connue ou quand le nombre de distributions possibles est réduit, et modifier en permanence les probabilités de présence des cartes dans les autres mains en fonction des renseignements que donnent les cartes jouées.

Considérons ce dernier point sur un exemple, pour bien montrer comment réaliser les modifications de probabilité. Supposons que l'ordinateur (en Nord) est le déclarant, qu'il a la main et décide de jouer le 4 de Pique alors que les Piques qu'il connaît sont répartis

---

54 Un honneur au bridge est soit un As, soit un Roi, soit une Dame, soit un Valet. On compte les points en prenant comme valeur As : 4, Roi : 3, Dame : 2, Valet : 1. Les points de distribution sont : 3 pour une chicane, 2 pour un singleton, 1 pour un doubleton.

55 SA : Sans-Atout. 1SA : enchère de 1 Sans-Atout.

56 Pas exactement, mais nous ne rentrerons pas dans trop de détails.

57 Cela peut paraître curieux, mais c'est ainsi.

de la façon suivante (Pique n'a pas encore été joué, et comme Nord est le déclarant, Sud est le mort) :

Nord : As, Roi,4 ;

Sud : 9,7,2.

Supposons qu'Est fournisse le 5, Sud le 9, qu'Ouest fasse le pli avec la Dame de Pique. Quelles informations pouvons-nous en retirer ? Sur Est peu de chose ; en revanche, nous pouvons améliorer nos connaissances sur Ouest. Comme nous possédons le Roi et l'As, nous savons qu'Ouest devait jouer le 10, le Valet ou la Dame pour réaliser la levée. S'il avait eu la Dame et le 10 sans le Valet, il aurait certainement joué le 10. Donc, nous savons qu'il a une des trois[58] distributions suivantes : (Dame, Valet, 10) ou (Dame, Valet) ou (Dame). Si nous ne disposons d'aucune information venant de l'ouverture concernant Pique, nous pensons que chaque carte non localisée a une chance sur deux de se trouver en Ouest. Donc la probabilité de chaque distribution est :

$$\begin{array}{lll} \text{Dame,Valet,10 :} & 0.5 \times 0.5 \times 0.5 & = 0.125 \\ \text{Dame,Valet :} & 0.5 \times 0.5 \times (1 - 0.5) & = 0.125 \\ \text{Dame :} & 0.5 \times (1 - 0.5) \times (1 - 0.5) & = 0.125 \end{array}$$

D'après le théorème de Bayes, nous savons que la probabilité que la Dame provienne d'une de ces distributions est :

$$\begin{array}{lll} \text{Dame,Valet,10 :} & 0.125/(0.125 + 0.125 + 0.125) & = 1/3 \\ \text{Dame,Valet :} & 0.125/(0.125 + 0.125 + 0.125) & = 1/3 \\ \text{Dame :} & 0.125/(0.125 + 0.125 + 0.125) & = 1/3 \end{array}$$

D'après ces résultats, nous voyons que la probabilité qu'Ouest ait le 10 est 0.33 et celle qu'il ait le Valet de 0.67. Nous devons donc ajuster les probabilités dans le jeu d'Ouest ainsi que dans le jeu d'Est. Nous venons de voir, sur cet exemple, que le re-calcul des probabilités n'est pas chose totalement triviale, même sur un cas simple.

Il nous reste maintenant à examiner comment jouer effectivement. À partir du moment où nous avons localisé l'intégralité des cartes, ou tout au moins que nous n'avons plus qu'un nombre de distributions possibles réduit, nous pouvons lancer une recherche exhaustive sur l'arbre de jeux. La fonction d'évaluation devra prendre en compte, non pas uniquement le nombre de plis faits, mais également le gain ou la perte au score. Il faudra aussi, dans certains cas, ne pas hésiter à utiliser la méthode de Michie telle que nous l'avons décrite, si, par exemple, nous devons, pour réaliser notre contrat, compter sur une faute adverse.

En début de partie, il faudra, si l'on est le déclarant, établir un plan de jeu. Réaliser un plan de jeu passable est relativement simple, on peut par exemple utiliser l'algorithme suivant :

- si le contrat est faisable avec les levées sur table, les jouer ;
- sinon, tenter d'affranchir une couleur longue ;
- si cela ne suffit pas, examiner la possibilité d'utiliser des coupes croisées ;

---

58 Il peut, bien sûr, posséder d'autres cartes à Pique, mais nous ne nous intéressons qu'à ces trois cartes-là : Dame, Valet, 10.

– si cela ne suffit toujours pas, calculer les impasses faisables, et si elles permettent de réaliser le contrat, les tenter.

Il est bien évident qu'un semblable algorithme serait complètement insuffisant pour réaliser un programme véritablement fort.

En ce qui concerne le jeu « au jour le jour », on peut implanter un certain nombre d'heuristiques classiques :

– en second jouer la plus faible carte ;
– en troisième jouer la plus forte carte ;
– à Sans Atout, entamer de la quatrième de la longue ;
– ...

Ces heuristiques ne permettront pas de réaliser un programme très fort mais doivent suffire à écrire un programme médiocre. On doit également utiliser les probabilités de répartition des cartes lorsque cela est possible.

Il existe bien d'autres heuristiques que l'on peut implanter, mais on ne possède que peu de résultats sur leur efficacité. Il existe quelques machines commerciales disponibles, mais elles sont toutes très faibles, même pas du niveau d'un joueur de club moyen.

15.9 Autres jeux

D'autres jeux ont été programmés avec plus ou moins de succès :

Le backgammon : Sans doute le domaine où a été obtenu le résultat le plus probant. Le programme BKG de Hans Berliner (qui faisait là son galop d'essai avant de se concentrer complètement sur les échecs) a en effet battu le champion du monde de Backgammon. On peut lire pour plus de renseignements (Berliner 1980). Berliner reconnaît cependant que le programme a eu de la chance lors de son match et il n'accordait à son programme que 25% de chances de gagner.

Les dames : On ne connaît pas de programme de dames françaises. En revanche, il existe de nombreux programmes pour les dames américaines[59] (checkers) dont le premier fut celui d'Arthur Samuel. Le niveau de ces programmes est extrêmement élevé. Le programme actuellement le plus fort est CHINOOK (Jonathan Schaeffer, Joseph Culberson, Norman Treloar, Brent Knight, Paul Lu and Duane Szafron (University of Alberta, Edmonton, Canada)). Au mois d'août 90, CHINOOK a participé et remporté l'Open du Mississipi en gagnant 7 parties, en annulant 7 et n'en perdant aucune. Il battit à cette occasion 2 des dix meilleurs joueurs mondiaux. Il participa ensuite à l'Open des États-Unis, où étaient présents neuf des dix meilleurs joueurs du monde dont le champion du monde Marion Tinsley[60]. Le tournoi se déroulait en huit matches de quatre parties chacun. CHINOOK en gagna quatre, fit quatre nulles et termina ainsi second du tournoi derrière Tinsley (+5, =3, -0). Le match contre Tinsley se termina par une nulle (+0, =4, -0). CHINOOK est ainsi le premier ordinateur à se qualifier pour une finale officielle de championnat du monde. La fédération américaine ayant refusé que le programme soit désigné comme challenger

59 La différence principale vient de la taille du damier (8x8) au lieu de (10x10), et à certaines règles de prise.

60 Le Dr. Marion Tinsley fut le meilleur joueur de checkers pendant près de 40 ans et durant cette période, où il joua des milliers de parties, il n'en perdit qu'une dizaine.

officiel, Tinsley a abandonné son titre et a affronté CHINOOK dans un match qui pouvait être considéré comme un championnat du monde officieux, à Londres, au mois d'août 1992. Tinsley l'emporta au terme des 40 parties sur le score de 4 parties gagnées, 2 parties perdues et 34 parties nulles, une performance remarquable pour CHINOOK. Un match revanche eut lieu en 1994, qui vit la victoire de CHINOOK par abandon, Marion Tinsley étant déjà affaibli par la maladie qui devait l'emporter. Depuis, CHINOOK a affronté le successeur de Tinsley, Don Lafferty, et a successivement annulé le premier match en 1994 et gagné le second en 1995. Il faut noter que la fédération de checkers autorise désormais chinook à participer à la majorité des grands tournois, et a créé un titre spécial de champion du monde homme-machine qui sansctionne le vainqueur du match opposant le champion du monde humain au meilleur programme.

Les principes généraux pour la programmation du jeu de Checkers sont les mêmes que pour le jeu d'échecs avec quelques différences notables :

- Le facteur de branchement est très inférieur au facteur de branchement des échecs.
- Les ouvertures sont un point fondamental. Il est possible de jouer un coup apparemment « évident » qui mène à une défaite en vingt coups. L'ouverture est donc une phase cruciale du jeu. Plutôt que de fournir à la machine une bibliothèque d'ouvertures standard, ce qui implique un travail long et fastidieux, l'équipe de CHINOOK a préféré lui fournir une *anti-bibliothèque*. Il s'agit, non pas de coups à jouer, mais de positions à éviter. L'anti-bibliothèque de CHINOOK contient seulement 2000 positions. Lors d'un match exhibition contre Don Lafferty la technique de l'anti-bibliothèque a permis au programme de trouver des innovations qui ont abouti à deux victoires, et elle lui a permis d'éviter trois défaites.
- Les finales et les bases de données permettant de les traiter sont fondamentales. Alors qu'aux échecs les parties atteignent rarement ce stade, la plupart des parties de checkers se concluent par des positions classiques connues de tous les joueurs et où la moindre déviation par rapport à la séquence standard peut transformer une victoire potentielle en nul, ou un nul en défaite. CHINOOK a une base de données résolvant tous les problèmes à 6 pièces (ou moins). Une base de données pour les positions à 7 pions est en cours de construction. Elle contiendra 35×10^9 positions.

Les gens intéressés par CHINOOK et la programmation des dames américaines peuvent se reporter aux excellents articles de Jonathan Schaeffer (Schaeffer 1989a; 1989b; Schaeffer *et al.* 1991; Schaeffer 1991; Schaeffer *et al.* 1992).

Jonathan Schaeffer estime que, dans l'état actuel de la technologie, il doit être possible de résoudre complètement le jeu de checkers (Schaeffer *et al.* 1992), c'est-à-dire de construire un programme capable de jouer parfaitement de façon certaine. Si l'équipe de CHINOOK y parvient, les checkers seront le premier jeu réellement complexe[61] à être résolu.

Le Go : Le Go fait partie des jeux extrêmement difficiles à programmer, bien qu'il soit à information complète. En effet, l'arbre de jeu ne peut pas être examiné de façon complète, même sur deux niveaux, en raison du très grand nombre possible de

61 Le tic-tac-toe est un jeu résolu, mais il n'est pas franchement complexe...

coups à chaque tour de jeu (plus de 300 en début de partie). D'autre part, le jeu est avant tout lié à des concepts positionnels stratégiques très difficiles à formaliser simplement. Le Go est très étudié en ce moment. Il s'agit probablement du prochain grand défi après les échecs. On peut lire avec profit (Chen *et al.* 1990) et (Shirayanagi 1990) sur le sujet.

Un des meilleurs programmes de Go est MFGO (Many Faces of Go), de David Fotland. Il a une force approximative de 13 Kyu[62], ce qui le place loin des très bons joueurs.

Dans un programme de Go, il y a trois parties principales : les structures qui représentent la connaissance, la fonction d'évaluation pour déterminer le score et la fonction de sélection des coups.

La représentation des connaissances : Il faut organiser la connaissance de façon à pouvoir l'utiliser efficacement et rapidement. Les structures ne sont pas relatives à des cases, ni globales, mais sont attachées à des groupes, des chaînes de pierres, des connexions ou des yeux.

L'évaluation : La fonction d'évaluation tente de donner à chaque groupe une valeur heuristique en terme de vie ou de mort. Un groupe ayant deux yeux sera inconditionnellement vivant et aura la plus forte évaluation, un groupe pouvant être capturé en peu de coups sera inconditionnellement mort. Entre ces deux extrêmes, David Fotland distingue 23 catégories en fonction des yeux potentiels, de la possibilité de s'échapper, le nombre de groupes voisins faibles... Cette fonction d'évaluation utilise un programme de recherche tactique pour évaluer les prises possibles.

La sélection de coups : Plutôt que de tester chaque coup, le programme utilise un ensemble de règles expertes qui lui suggèrent des coups à essayer en priorité, en fonction de certains critères. On vérifie, après la réalisation du coup suggéré, qu'il a bien mené au but recherché. Par exemple, si une règle offensive suggère d'attaquer un groupe, le groupe doit être plus faible après que le coup ait été réalisé.

Cette rapide description montre bien que l'approche de la programmation du Go se distingue très nettement de celle des autres jeux. Elle s'appuie plus sur des considérations expertes que sur des considérations de force brute. Il reste à savoir quel sera l'avenir de ces programmes. Les premiers programmes d'échecs étaient eux aussi des programmes plus orientés vers l'expertise que vers la force, et la tendance s'est lentement inversée.

---

62 Le classement des joueurs de Go est complexe. Les joueurs « faibles » sont classés en Kyu. Plus faible est le Kyu, plus fort est le joueur. Les débutants ont une force de 30 Kyu approximativement. Un joueur pratiquant régulièrement en club, pendant un ou deux ans, doit atteindre une force de l'ordre de 10 Kyu. Lors d'une partie, la différence des Kyu des deux joueurs est utilisée pour déterminer le nombre de pierres de handicap. Arrivé à 1 Kyu, le rang suivant est 1 Dan, et le rang le plus élevé possible est 9 Dan. Les meilleurs joueurs amateurs ont une force de 7 Dan environ. David Fotland, l'auteur de MFGO, a une force de 3 Dan.

15.10 Les jeux vus par Conway

La fin de ce chapitre sera consacrée à une théorie assez récente (à l'échelle des mathématiques) et qui permet de rapprocher deux domaines chers aux informaticiens : jeux et nombres. Nous nous limiterons à une présentation très succincte et informelle de cette théorie, en nous focalisant presque exclusivement sur les jeux. Nous invitons le lecteur à se plonger dans les délices des deux documents consacrés aux *jeux de Conway* :

- la première référence, couramment désignée sous l'acronyme ONAG (On Numbers and Games (Conway 1976)) est la plus formelle et est consacrée de façon équitable aux nombres et aux jeux...
- la seconde (Conway *et al.* 1982), nettement plus ludique, est plus particulièrement consacrée aux jeux et est une source inépuisable de plaisirs, et d'exercices sur les jeux à zéro, un ou deux joueurs.

15.10.1 Quels jeux ?

Les jeux auxquels nous nous intéresserons ici doivent respecter un ensemble de conditions assez sévères :

- ils opposent deux joueurs, que nous appellerons Gauche (couleur blanche) et Droite (couleur noire) ;
- ils comportent un nombre habituellement fini de positions, dont une position initiale ;
- dans chaque position, des règles définissent précisément les coups que peut effectuer chacun des joueurs pour aboutir à une nouvelle position (les options du joueur) ;
- Gauche et Droite jouent alternativement (sans que soit précisé qui commence) ;
- le jeu est à information complète et ne fait pas intervenir le hasard ;
- c'est le premier joueur qui ne peut plus jouer (il n'a plus d'options) qui perd (on parle de jeu normal, par opposition aux jeux de misère) ;
- le jeu doit se terminer par la défaite d'un des deux joueurs.

Ces conditions sont (pour la plupart) nécessaires à l'application saine des algorithmes que nous avons étudiés précédemment. Le point nouveau est la condition de « jeu normal » (c'est le dernier qui joue qui gagne) . Cette qualité est absente de bon nombre de jeux courants (échecs, Othello, Dames) mais n'a pas l'aspect restrictif qu'on pourrait lui donner de prime abord : la théorie a déjà été partiellement étendue aux jeux de misère et les règles de certains jeux peuvent être modifiées sans répercussions majeures pour respecter cette condition[63].

Un jeu qui respecte ces conditions est en fait entièrement défini par une position initiale et par la donnée pour chaque position, de toutes les positions que peut atteindre chacun des joueurs (ses options) à partir de cette position. Il peut donc se représenter sous la forme d'un arbre dont la racine correspond à la position initiale, et dont chaque arc peut être de deux types : gauche et droite. Chaque nœud correspond à une position et comporte deux ensembles de fils : les fils gauches (resp. droits) représentent les jeux

63 L'Othello par exemple. Une fois toutes les cases occupées, le jeu rentre dans une seconde phase dans laquelle chacun retire un pion de sa couleur à son tour. Le premier qui ne peut plus jouer a perdu. Le seul cas où le gagnant de ce jeu n'est pas celui de l'Othello est le nul (32 pions partout).

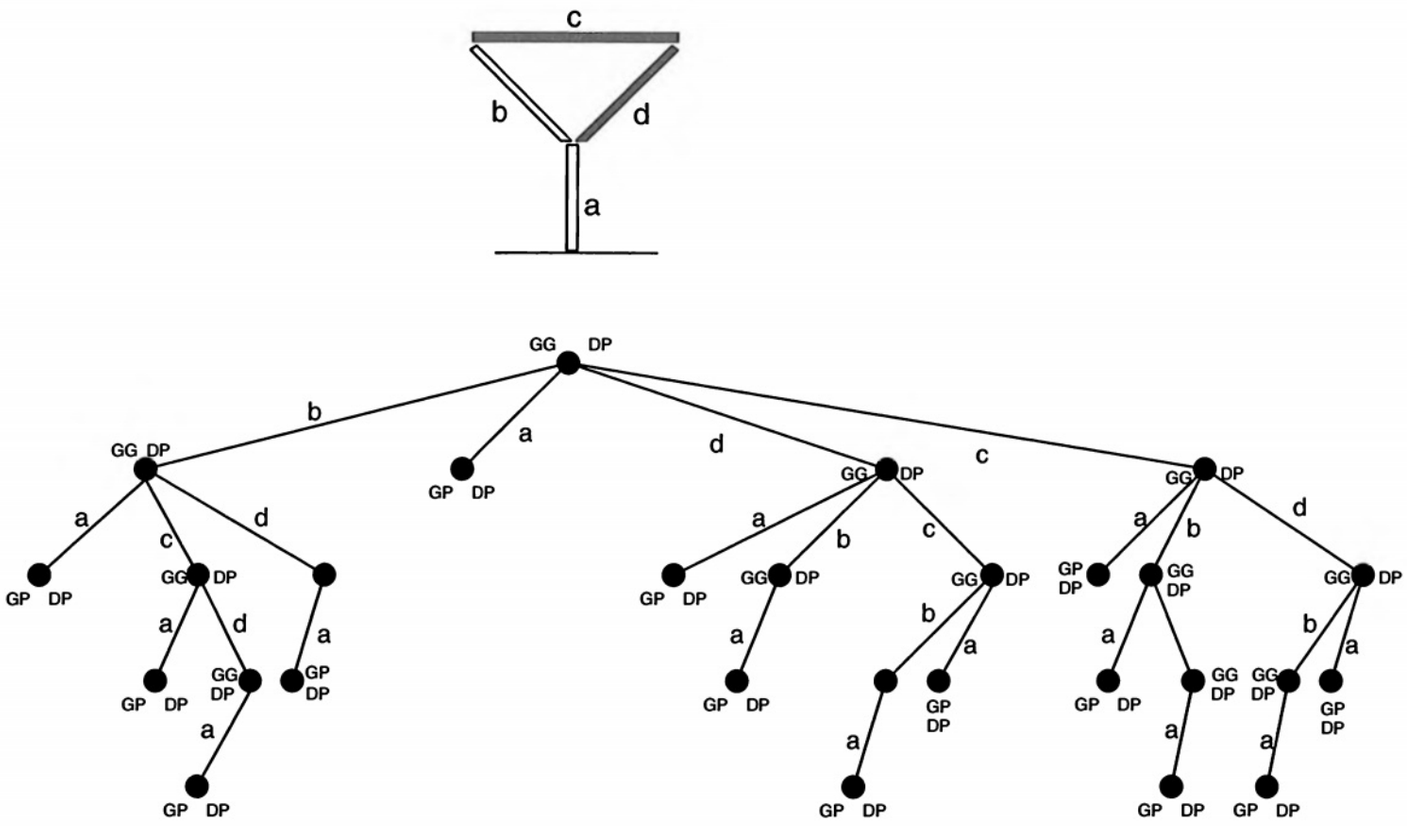


Figure 15.18 – Une position pour le jeu de Hackenbush

que Gauche (resp. Droite) peut atteindre en jouant un coup dans cette position. Nous distinguerons toujours par la suite les jeux (réels, définis par une position initiale et des règles) des *jeux de Conway* :

Définition 15.3 – *Un* jeu de Conway x *noté* $x = \{G|D\}$ *est formé à partir de deux ensembles* G *et* D *(éventuellement vides) de* jeux de Conway. *Tous les jeux de Conway sont formés de cette façon.*

Notation : Étant donné un jeu $x = \{G|D\}$, où $G = \{x_1^G, \ldots, x_n^G\}$ et $D = \{x_1^D, \ldots, x_m^D\}$, nous le noterons alternativement :

$$x = \{x_1^G, \ldots, x_n^G | x_1^D, \ldots, x_m^D\}$$

ou plus simplement

$$x = \{x^G | x^D\}$$

Le caractère visiblement inductif de cette définition, dont la base est fournie par l'ensemble vide, fait bien apparaître la nature « arborescente » d'un jeu de Conway. Afin d'éclaircir un peu les choses, considérons le jeu de Hackenbush. Ce jeu se joue sur un ensemble d'arêtes bicolores qui n'est pas fixé (un exemple de position initiale est donné figure 15.18). Gau**che** peut supprimer n'importe quelle arête de couleur Blan**che** sur cette figure. Toutes les arêtes (sans considération de couleur) qui ne sont plus connectées au sol seront alors simultanément supprimées. Dr**oi**te peut faire la même chose avec les arêtes N**oi**res. Le premier qui ne peut plus jouer a perdu. La représentation arborescente du jeu de Conway correspondant à cette position initiale est donnée dans la même figure.

Un « jeu réel » implicite est associé à chaque jeu de Conway. Gauche (resp. Droite) peut jouer en choisissant un des fils Gauche (resp. Droite) du nœud racine du jeu de

Conway, le nœud choisi devient le nouveau jeu de Conway... Le premier qui n'a plus de choix a perdu[64].

Une fois construit le jeu de Conway qui correspond à un jeu, il est simple de déterminer qui peut (contre toute défense de son adversaire) gagner. Il suffit d'étiqueter (en partant des feuilles de l'arbre *i.e.*, des nœuds ayant au moins un ensemble de fils vide) chaque nœud de la façon suivante :

- si tous les fils gauches du nœud sont étiquetés GD (gagnant pour Droite) alors étiqueter le nœud PG (perdant pour Gauche) sinon, l'étiqueter GG (gagnant pour Gauche). En effet, si toutes les options de Gauche sont des victoires pour Droite, Gauche perd, sinon, il gagne en jouant l'option portant l'étiquette PD (perdant pour Droite) ;
- si tous les fils droits du nœud sont étiquetés GG (gagnant pour Gauche) alors étiqueter le nœud PD (perdant pour Droite) sinon, l'étiqueter GD (gagnant pour Droite). En effet, si toutes les options de Droite sont des victoires pour Gauche, Droite perd, sinon, il gagne en jouant l'option portant l'étiquette PG (perdant pour Gauche).

Les bases du processus d'étiquetage sont les nœuds ayant un des ensembles de fils Gauche (resp. Droite) vide. Dans ce cas, on sait que tous les fils Gauche (il n'y en a pas) (resp. Droite) du nœud sont étiquetés GD (resp. GG), et on peut donc étiqueter le nœud PG (resp PD).

À la fin du processus, chaque nœud portera deux mentions : (PG ou GG) et (PD ou GD). Si l'on considère un nœud donné, les quatre cas se résument de la façon suivante :

	PG	GG
PD	Le joueur qui commence perd, nous dirons que le jeu est NUL (0)	Le joueur Gauche gagne à tous les coups, nous dirons que le jeu est POSITIF (+)
GD	Le joueur Droite gagne à tous les coups, nous dirons que le jeu est NÉGATIF (−)	Le joueur qui commence gagne, nous dirons que le jeu est FLOU (||)

Les exemples qui suivent serviront de justification à ce classement aux consonances « numériques ».

15.10.2 Quelques jeux de Conway

La figure 15.19 présente quelques positions du jeu de HackenBush et leur jeu de Conway associé, représenté sous forme arborescente. Le plus simple d'entre eux est le jeu dont la position initiale ne contient aucune arête et que nous appellerons 0. Il s'agit bien évidemment d'un jeu nul, puisque ni Gauche ni Droite ne peuvent jouer.

---

64 De façon traditionnelle en théorie des graphes, on définit de façon analogue le *jeu de Nim* sur un arbre (ou un graphe) : le premier joueur choisit un des nœuds fils du nœud racine, le second choisit un des fils du nœud choisi par son adversaire... Le premier qui n'a plus de choix a perdu. On verra que les jeux de Nim ne sont qu'un cas particulier des jeux de Conway : ce sont les jeux impartiaux.

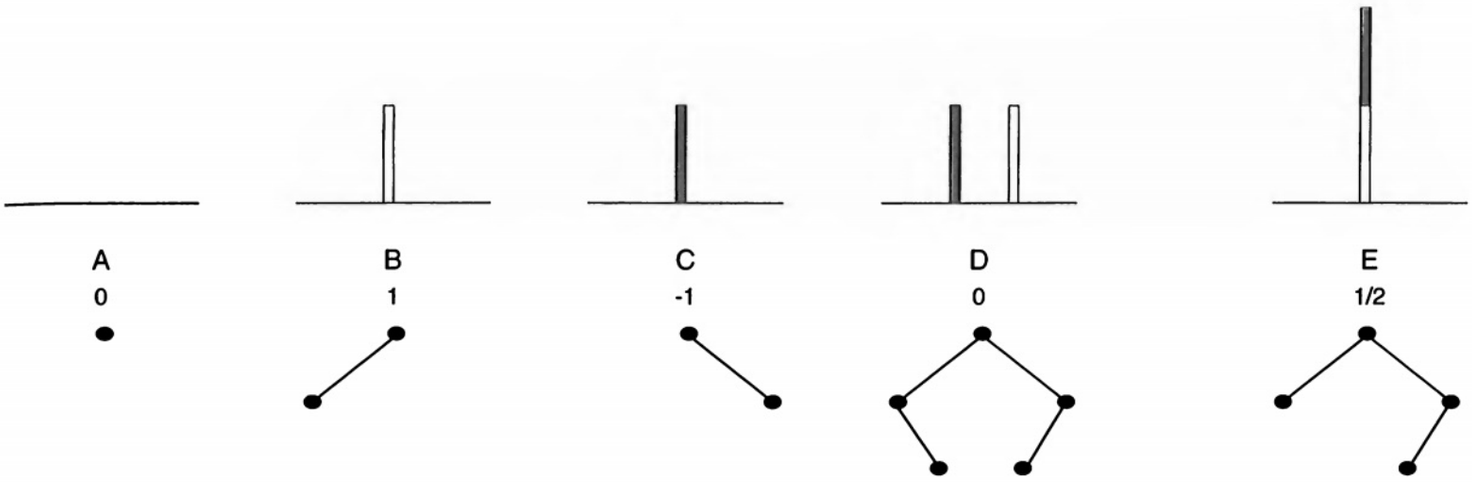


Figure 15.19 – Quelques positions pour le jeu de Hackenbush

Le jeu défini par la position B présente un avantage d'un coup réservé pour Gauche. Il s'agit d'un jeu positif (GG : Gagnant pour Gauche et PD : Perdant pour Droite). Nous l'appellerons 1. Le jeu C dans lequel les couleurs Blanche et Noire ont été inversées est négatif et donne un avantage d'un coup à Droite. Nous l'appellerons -1.

De manière générale, soit un jeu de Conway $x = \{x_1^G, \ldots, x_n^G | x_1^D, \ldots, x_m^D\}$, le jeu opposé de x, noté $-x$ est le jeu où, pour toute position, les options de Gauche et Droite ont été inversées (sur la représentation arborescente cela revient à effectuer une symétrie d'axe vertical). Formellement :

$$-x = \{-x_1^D, \ldots, -x_m^D | -x_1^G, \ldots, -x_n^G\}$$

Considérons maintenant le jeu défini par la position D, formé à partir de 1 et de -1. Si Gauche commence, il perd puisque Droite peut répondre en laissant Gauche dans un jeu nul (déjà vu). Si Droite commence, il perd aussi puisque Gauche peut laisser Droite dans le même jeu nul. Le jeu formé à partir des jeux 1 et -1 est donc un jeu nul. Autrement dit :

$$1 + (-1) = 0$$

Si l'addition de deux jeux de Hackenbush est simple à appréhender, comment la définir dans le cadre plus général des jeux de Conway ? Considérons deux jeux de Conway x et y. Le jeu $x + y$ sera le jeu dans lequel chaque joueur peut, à chaque fois que c'est son tour, décider de jouer dans x ou dans y. Naturellement, c'est le premier qui ne peut plus jouer qui a perdu !

Dans le jeu $x + y$, Gauche peut donc décider de jouer dans x et le jeu obtenu sera la somme de l'option gauche de x choisie et de y, ou décider de jouer dans y et dans ce cas, le jeu obtenu sera la somme de l'option gauche de y et de x. Le même raisonnement s'appliquant à Droite, on obtient :

$$x + y = \{(\cup_i(x_i^G + y)) \cup (\cup_i(x + y_i^G)) | (\cup_i(x_i^D + y)) \cup (\cup_i(x + y_i^D))\}$$

ou, en employant une notation certes un peu abusive, mais bien plus agréable :

$$x + y = \{x^G + y, x + y^G | x^D + y, x + y^D\}$$

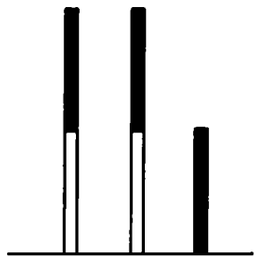


Figure 15.20 – Un jeu nul ?

Il est simple de vérifier que le jeu défini par la position A, et que nous avons appelé 0, est bien un élément neutre pour $+$. La démonstration se fait, bien sûr, par induction.

Soit x un jeu de Conway, on veut montrer que $x + 0 = x$. Supposons la propriété vraie pour toute option (Gauche ou Droite) de x. On a :

$$x + 0 = \{x^G + 0 | x^D + 0\} = \{x^G | x^D\} = x$$

C.Q.F.D.

Continuons notre exploration en étudiant le jeu défini par la position E. Si Gauche commence, il gagne en « cueillant » son unique arête. Si Droite commence, Gauche peut encore jouer et laisse Droite sur un jeu nul. Le jeu est donc positif. Quel nom pourrait-on lui donner ?

Considérons le jeu défini par la somme du jeu défini par la position E et de (-1) et étudions son « signe ». Dans ce jeu, si Gauche commence, Droite peut répondre dans -1 et Gauche perd. Si Droite commence (dans E), Gauche répond (dans E) et Droite gagne en répondant dans -1. Le jeu défini est donc négatif. Autrement dit, le jeu défini par E est compris entre 0 et 1. Nous l'appellerons $\frac{1}{2}$.

Pour justifier son nom, nous allons tout simplement considérer le jeu $\frac{1}{2} + \frac{1}{2} + (-1)$ (Cf. figure 15.20) et montrer qu'il est nul :

- si Gauche commence, il joue dans un des $\frac{1}{2}$, Droite répond dans l'autre $\frac{1}{2}$, Gauche répond dans le même $\frac{1}{2}$ et Droite gagne en jouant dans -1 ;
- si Droite commence, il joue de préférence dans un des $\frac{1}{2}$. Gauche répond dans l'autre $\frac{1}{2}$, Droite est alors obligé de prendre son -1 et Gauche gagne en prenant le dernier arc (du premier $\frac{1}{2}$).

Le jeu est donc nul, et le nom de $\frac{1}{2}$ justifié !

De façon plus générale, comment comparer un jeu x à un jeu y ?

$$\text{On a :} \left| \begin{array}{l} y - x \geq 0 \Leftrightarrow \\ y + (-x) \geq 0 \Leftrightarrow \\ \{y^G | y^D\} + \{-x^D | -x^G\} \geq 0 \Leftrightarrow \\ \{y^G + (-x), y + (-x^D) | y^D + (-x), y + (-x^G)\} \geq 0. \end{array} \right.$$

Si ce dernier jeu est *positif* ou *nul*, cela signifie que Droite perd quand il commence. Il ne doit donc y avoir aucune option droite de ce jeu qui mène Droite à la victoire quand

il commence : ni $y^D + (-x)$, ni $y + (-x^G)$ ne doivent être inférieur ou égal à 0. On aboutit donc à la définition suivante :

Définition 15.4 – *Étant donnés deux jeux de Conway, x et y, on dira que $x \leq y$ ssi :*
- *il n'existe aucun y^D tel que $y^D \leq x$ et*
- *il n'existe aucun x^G tel que $y \leq x^G$.*

Définition 15.5 – *Étant donnés deux jeux de Conway, x et y, on dira que $x = y$ ssi $x \leq y$ et $y \leq x$.*

L'égalité entre deux jeux est donc une relation *définie* qui est réflexive, symétrique (par construction) et transitive. C'est donc une relation d'équivalence. Nous pourrons dire dorénavant qu'un jeu a une certaine valeur a s'il appartient à la même classe d'équivalence qu'un jeu que nous avons appelé a. En particulier, un jeu est nul (ou vaut 0) s'il est égal au jeu de Conway correspondant à la position de Hackenbush A.

15.10.3 Les nombres

En fait, tous les jeux de Conway que nous avons pu observer jusqu'ici font partie d'une sous-classe très particulière des jeux de Conway, que nous appellerons nombres de Conway, et qui sont définis de la façon suivante :

Définition 15.6 – *Un nombre de Conway $x = \{G|D\}$ est un jeu de Conway, formé de deux ensembles (éventuellement vides) G et D de nombres de Conway, et tel qu'il n'existe aucun élément de D qui soit $\leq$ à un élément de G.*

La relation $\geq$ telle qu'elle a été définie sur les jeux de Conway est totale sur les nombres de Conway.

Au-delà de l'addition, de la négation, il est possible de définir multiplication, division, etc. sur les nombres de Conway, mais ces définitions perdent toutes leurs « bonnes » propriétés quand elles sont étendues aux jeux de Conway.

La structure ainsi formée est une structure de corps totalement ordonné[65]. Nous ne reparlerons plus par la suite des nombres de Conway, mais uniquement des jeux de Conway.

15.10.4 Les jeux impartiaux et les nimbres

Le jeu de Hackenbush que nous venons de voir est un jeu partisan (chaque joueur a ses options « réservées »). Dans cette partie, nous nous limiterons à la classe des jeux dits *impartiaux* dans lesquels toute option de Gauche est aussi une option de Droite et vice-versa, ceci pour toute position du jeu. La représentation arborescente d'un jeu impartial est bien sûr symétrique par rapport à un axe vertical et son opposé n'est autre que lui-même !

Considérons par exemple le jeu de Hackenbush Gris dans lequel toutes les arêtes (grises) peuvent être enlevées par Gauche ou Droite indifféremment. Il s'agit évidemment d'un jeu impartial. Quelques positions du jeux de Hackenbush Gris sont présentées dans la figure 15.21.

---

65 Le lecteur intéressé pourra se reporter à (Conway 1976), pages 17–21.

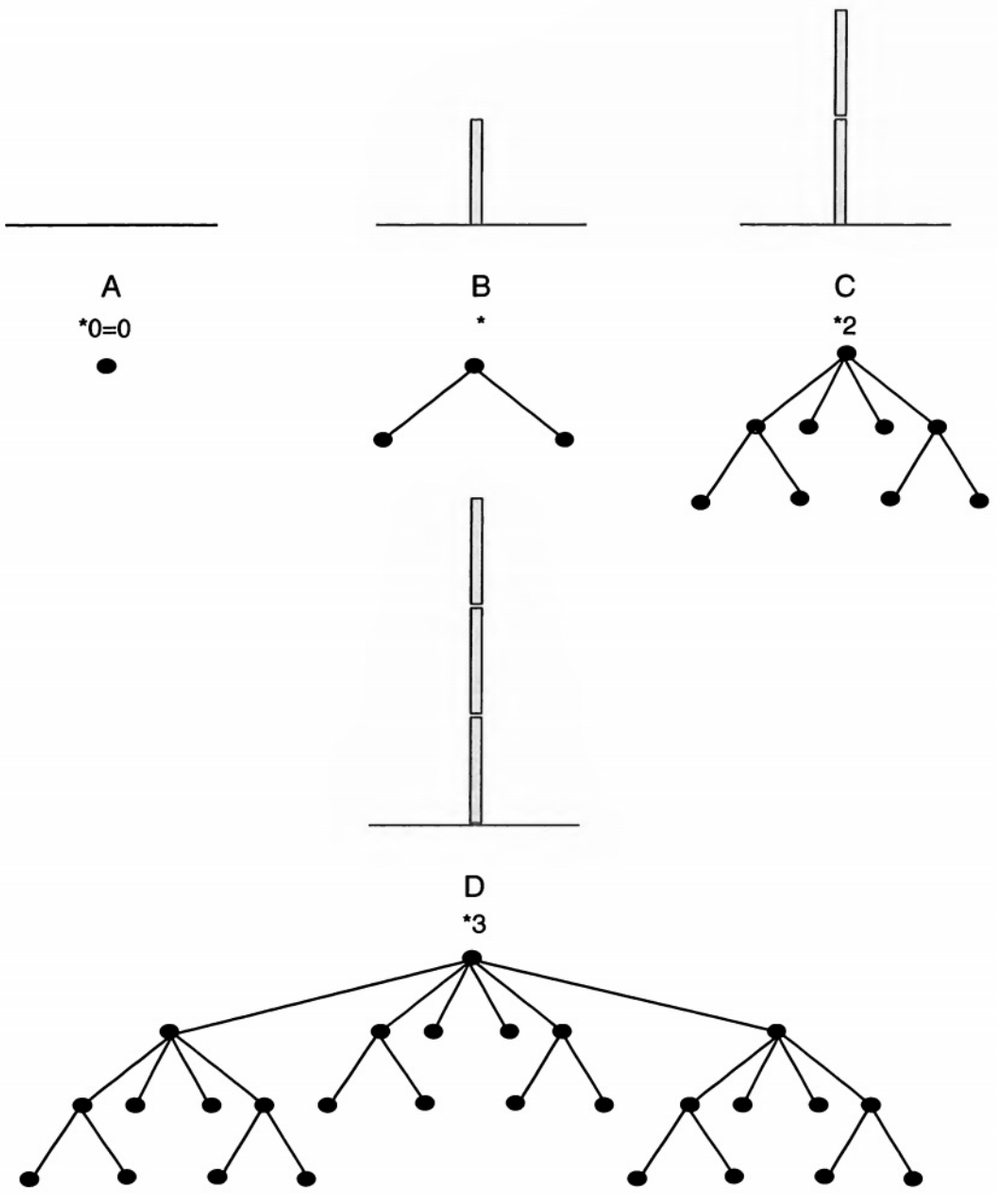


Figure 15.21 – Quelques positions de Hackenbush Gris

Comme précédemment, le jeu défini par la position A est nul. Dans le jeu défini par B, si Gauche commence, Droite perd et si Droite commence, Gauche perd. Il s'agit donc de notre premier jeu flou que nous noterons $*$. Ce jeu de Conway (le lecteur pourra facilement démontrer qu'il ne s'agit pas d'un nombre de Conway) a des propriétés « arithmétiques » remarquables puisque l'on peut facilement montrer qu'il est plus petit que tout *nombre de Conway* strictement positif, plus grand que tout *nombre de Conway* strictement négatif, et cependant différent de 0 ! En fait, tous les jeux flous (ce ne sont pas des nombres de Conway, rappelons-le) ont des propriétés souvent étranges, parfois choquantes.

La position suivante C possède deux options gauches : 0 et $*$ (et, bien sûr, les deux mêmes options pour Droite). Nous noterons le jeu ainsi défini $*2 = \{0, *|0, *\}$. Plus généralement, le jeu de Hackenbush gris défini par une position initiale prenant la forme d'une tige de longueur n sera égal à :

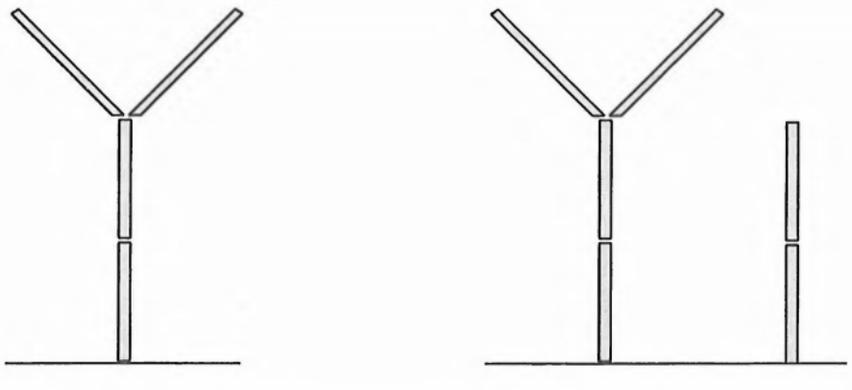


Figure 15.22 – Quelques positions plus complexes

Il est facile de montrer que tous les $*n$ ($n > 0$) sont des jeux flous. Nous les appellerons des *nimbres*[66].

Dans des positions de Hackenbush Gris un peu plus complexes (telle la position E), la valeur obtenue peut-elle s'exprimer en termes de nimbres ? Dans le cas de E, on a :

$$\mathsf{E} = \{*3, *1, *0 | *3, *1, *0\}$$

Le jeu défini par E a-t-il pour valeur un nimbre ? Et si oui, lequel ? La réponse est contenue dans le théorème suivant :

Théorème 15.1 – *(dit du plus petit exclus ou ppex) Soit*

$$x = \{*a, *b, \ldots, *c | *a, *b, \ldots, *c\}$$

un jeu de Conway impartial, alors la valeur de x est $*p$ où p est le plus petit entier positif non nul qui ne fait pas partie de l'ensemble $\{a, b, \ldots, c\}$. On dira que $*p$ est le ppex de $*a, *b, \ldots, *c$.

Le jeu défini par E vaut donc $*2$. On peut le vérifier en l'additionnant à l'opposé de $*2$ qui n'est autre que $*2$ lui-même. Le jeu obtenu est le jeu dont la position initiale est la position F.

---

66 D'après les *nimbers* de Conway, eux-mêmes inspirés du jeu de Nim, jeu impartial qui, historiquement, est à la base de la théorie des jeux impartiaux. Rappelons que le *jeu de Nim* a déjà été étudié dans le cadre de la théorie des graphes. L'ensemble des résultats de cette section ont donc été déjà établis sous une forme voisine en 1936 par R.P. Sprague et, indépendamment, en 1939, par P.M. Grundy. : l'équivalent d'un jeu sera la donnée d'un graphe (X, Γ) et du nœud correspondant à la position initiale i, jouer correspond à choisir un nœud dans $\Gamma(i)$ qui deviendra la position initiale pour le joueur adverse. Le premier qui ne peut plus jouer a perdu. La valeur d'un jeu sera le nombre de Grundy du nœud i : une fonction de Grundy est une fonction g de X dans N telle que $\forall x \in X, g(x)$ est égale au plus petit entier qui n'appartient pas à $\Gamma(x)$ (elle existe et est unique dans les cas qui nous intéressent). La somme de deux jeux (X_1, Γ_1), nœud initial i_1 et (X_2, Γ_2), nœud initial i_2 sera définie par le graphe somme des deux graphes (c'est le graphe $(X_1 \times X_2, \Gamma)$ où $\Gamma(x_1, x_2) = \cup_{y \in \Gamma_1(x_1)}\{(y, x_2)\} \cup_{z \in \Gamma_2(x_2)} \{(x_1, z)\}$) et la position initiale correspondante sera le nœud (i_1, i_2). Enfin, le nombre de Grundy de la somme de deux jeux est égal au ou-exclusif bit à bit des nombres de Grundy de chacune des positions initiales dans les graphes correspondants. Le lecteur intéressé pourra se reporter par exemple à (Berge 1970), chapitre 14.

- Si Gauche joue dans $*2$, Droite peut jouer dans E un coup identique. La somme de deux jeux impartiaux identiques étant nulle, Gauche perd.
- Si Gauche joue $*0$ ou $*1$ dans E, le même raisonnement s'applique et Gauche perd.
- Si Gauche joue $*3$ dans E, on est ramené à jouer dans $*3+*2$. Il suffit alors à Droite de jouer l'option $*2$ de $*3$ pour mettre Gauche dans la position $*2 + *2 = 0$. Gauche perd encore.

La somme est donc nulle et le jeu défini par E vaut bien $*2$.

Ce théorème a un corollaire important indiquant que tout jeu impartial

$$x = \{a, b, c, \ldots | a, b, c, \ldots\}$$

a forcément pour valeur un nimbre.

La démonstration inductive est simple : supposons que la propriété soit vraie pour toute option de x, alors x est de la forme $\{*i, *j, *k, \ldots | *i, *j, *k, \ldots\}$ puisque les jeux $a, b, c, \ldots$ sont des nimbres (par hypothèse). Une simple application du théorème du ppex permet d'aboutir au résultat final et fournit, de plus, la valeur du jeu (le ppex de $\{*i, *j, *k, \ldots\}$). □

Cette démonstration est aussi à la base d'une procédure de calcul récursive de la valeur de tout jeu impartial. Sa mise en œuvre informatique est aisée[67].

Le seul point qui reste à éclaircir est l'addition de deux nimbres. On sait déjà que $*n + *n = 0$, mais que dire de $*a + *b$ en général ? On démontre en fait que :

$$*a + *b = *(a \oplus b)$$

où $\oplus$ est le OU exclusif binaire, bit à bit. Ainsi, $*7 + *2 + *3$ vaut (en passant en base 2) $*(111 \oplus 010 \oplus 011) = *(110) = *6$.

Nous disposons maintenant de tous les outils pour jouer brillamment dans le jeu défini par la position initiale G de la figure 15.23. Ce jeu est la somme de trois jeux :

- Les deux derniers sont connus et valent $*4$ et $*2$.
- Le premier est égal à $\{*4, *2, *1, *0 | *4, *2, *1, *0\}$ et vaut donc $*3$ d'après le théorème du ppex.

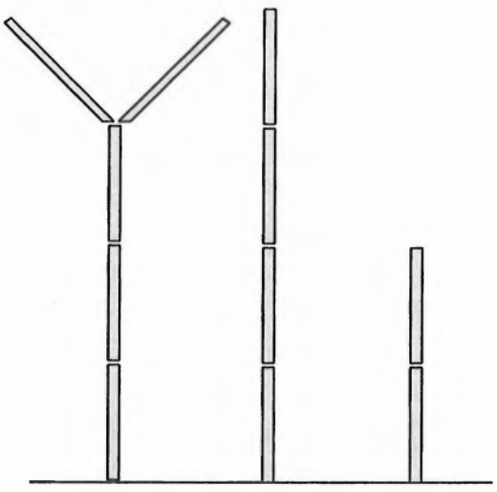
La somme des trois jeux vaut donc $*3 + *4 + *2 = *5$. Le jeu n'est pas nul, et nous pouvons donc gagner. Pour cela, il faut jouer un coup qui mettra l'adversaire dans une position nulle. On va donc jouer l'option $*1$ dans le jeu $*4$, le jeu obtenu vaut $*3 + *1 + *2 = *0 = 0$. Ce même principe pourra s'appliquer à la position résultante du coup de l'adversaire qui, si l'on ne commet pas d'erreur, est voué à l'échec.

15.10.5 Conclusion

La contribution de Conway ne s'arrête pas aux quelques types de jeux que nous avons pu évoquer. Le lecteur pourra se plonger dans les délices des jeux chauds (les plus palpitants), pour ensuite les refroidir juste assez pour tenter de discerner leur valeur moyenne, au milieu de la vapeur.

---

67 Nous l'avons ainsi appliquée au jeu de Chomp (impartial). De même, nous avons appliqué la théorie des jeux partiaux au jeu de Cross-Cram (partial). Il devient assez rapidement nécessaire de s'appuyer sur une méthode de « mémoïzation » pour éviter le calcul répété de la valeur d'une position (table de transposition s'appuyant sur des techniques d'arbre de discrimination ou de tables de hachage par exemple, cf. (Cormen *et al.* 1990)).



G

Figure 15.23 – Qui gagne ? Et comment ?

À notre connaissance, aucune application de cette théorie à des jeux difficiles n'a été réalisée[68] et si elle a été « présentée » c'est surtout pour son caractère original. Un premier pas, d'une complexité déjà très (trop ?) forte pourrait être de calculer le type (nul, flou, positif ou négatif) du jeu d'Othello dans sa position initiale.

68 Il a tout de même été possible de poursuivre grâce à elle et un codage informatique de qualité, l'analyse du jeu de « Sprouts » (Cf (Conway *et al.* 1982; Applegate *et al.* 1991)), jusqu'à une taille de 11 points initiaux alors que selon Gardner, Conway estimait en 1967 que l'analyse du jeu à 8 points était hors de portée des ordinateurs.

CHAPITRE 16

Les systèmes experts

Jean-Marc Alliot — Thomas Schiex

16.1 Introduction

Le terme *système expert* (SE) a fait son apparition au début des années 70. Les systèmes experts sont le résultat de l'évolution de l'IA et en particulier des désillusions des années 60 quant à l'efficacité des méthodes faibles pour la résolution générale de problèmes.

Le mécanisme de construction d'un système expert est proche de celui que nous avons décrit dans le chapitre consacré aux méthodes faibles : il faut dégager une méthode de représentation des connaissances et construire un ensemble de règles de production. Les différences portent principalement sur deux points :

- les systèmes experts s'appliquent généralement dans des domaines moins bien spécifiés. Il est en effet fort difficile de préciser totalement les principes de compréhension du langage naturel ou du diagnostic médical, alors que les règles du jeu d'échecs sont simples à formaliser ;
- la construction d'un système expert tente de tenir compte au mieux de la structure du problème lui-même pour dégager des techniques de représentation et de résolution appropriées, alors que les méthodes faibles sont des techniques générales de parcours d'arbres et de graphes.

En résumé, les systèmes experts s'appliquent dans des domaines restreints où on ne connaît pas d'algorithme général de résolution, où la connaissance est largement fragmentée et les techniques de résolution fortement liées au savoir-faire de l'expert humain, savoir-faire lui-même parcellaire et parfois incomplet, voire inconsistant.

Certains auteurs (Laurière 1986) exigent que le système, pour être dit *expert* soit aussi capable de dialoguer avec l'homme dans un langage simple. Cette caractéristique, si elle est souvent présente dans les systèmes experts, ne nous semble cependant pas liée à l'objet même du système expert.

Notons également que le concept que recouvre le terme *système expert* est un peu flou. Il suffit, pour s'en persuader, de lire différents ouvrages consacrés à l'IA : lorsque l'on compare les chapitres présentant les SE, on a parfois la sensation qu'ils ne parlent pas des mêmes objets et laissent souvent l'impression d'un indescriptible fouillis. La raison en est que l'on utilise pour écrire des systèmes experts diverses méthodes, et tenter de les

regrouper dans un chapitre relève bien souvent de la gageure[1]. Si le but à atteindre est bien le même, les techniques pour l'atteindre sont très différentes. Nous avons cependant choisi de suivre la voie commune, en insistant sur l'importance fondamentale de la logique pour la réalisation des SE[2]. Une vision plus complète des nombreuses logiques utilisables pour la représentation des connaissances se trouve dans (Turner 1986b).

Signalons enfin une approche différente des SE, introduite principalement par R. Davis dans (Davis 1982b), la programmation par modèles. Nous y reviendrons à la fin de ce chapitre.

Les systèmes experts s'inscrivent dans le cadre de ce que Feigenbaum a appelé *l'ingénierie cognitive*, ensemble de techniques et de méthodes permettant d'appréhender la connaissance humaine sous un aspect formel et de la traiter sous cette forme symbolique. Ce sont ces techniques que nous allons nous attacher à décrire rapidement dans ce chapitre.

16.2 La représentation des connaissances

La représentation des connaissances est un problème qui se pose à tout système expert, qu'il soit ou non à base de règles. Le choix d'une bonne représentation est fondamental et détermine toute la suite du développement du système.

16.2.1 Logique des prédicats

La logique des prédicats est un des moyens les plus courants et les plus commodes de représenter des faits ou des règles, dans la mesure où le domaine étudié est suffisamment élémentaire. Nous avons déjà largement discuté du calcul des prédicats, nous ne reviendrons pas dessus. Nous allons simplement présenter dans ce chapitre quelques exemples d'utilisation.

Supposons que nous souhaitions représenter la phrase : **F-GI135 est un avion**. Nous pouvons écrire :

$$\text{Avion(F-GI135)}$$

où $\text{Avion}(x)$ est un symbole de prédicat unaire et F-GI135 un symbole de constante. De même, il est aisé de représenter : **Tous les avions volent** par :

$$\forall x, \text{Avion}(x) \rightarrow \text{Vole}(x)$$

On peut même raffiner en disant **tous les avions qui ne sont pas en panne volent** :

$$\forall x, \text{Avion}(x) \wedge \neg\text{En\_Panne}(x) \rightarrow \text{Vole}(x)$$

On voit donc que le langage de la logique du premier ordre est relativement riche.

---

1 Certains auteurs ((Rich 1987) par exemple) se refusent d'ailleurs à consacrer un chapitre aux systèmes experts et préfèrent présenter séparément les différentes techniques.

2 Suivant en cela (Nilsson 1988) ou (Thaysse and others 1988).

Remarquons cependant qu'il existe plusieurs moyens de représenter les mêmes notions. Ainsi **F-GI135 est un avion** peut, certes, se représenter :

$$\text{Avion(F-GI135)}$$

Mais on peut également l'écrire :

$$\text{Est\_un(F-GI135, Avion)}$$

Le prédicat Est_un est un prédicat d'instanciation qui permet de transformer toute représentation par un *prédicat unaire* en une représentation par un *prédicat binaire.*

Considérons maintenant la phrase : **F-GI135 va de Paris à Londres**. On peut la représenter par le prédicat ternaire :

$$\text{Aller(F-GI135, Paris, Londres)}$$

Mais on peut décomposer ce prédicat ternaire en une conjonction de prédicats binaires :

$$\text{Avion(Aller, F-GI135)} \wedge \text{Origine(Aller, Paris)} \wedge \text{Destination(Aller, Londres)}$$

Nous avons simplement traduit la structure fonctionnelle du prédicat initial qui était :

$$\text{Aller(}\mathbf{Avion}\text{, }\mathbf{Origine}\text{, }\mathbf{Destination}\text{)}$$

De façon générale, tout prédicat n-aire a une structure fonctionnelle de la forme :

$$\text{Nom\_pred(fonc}_1\text{, fonc}_2\text{, ... fonc}_n\text{)}$$

et peut se représenter :

$$\text{fonc}_1\text{(Nom\_pred, val}_1\text{)} \wedge \text{fonc}_2\text{(Nom\_pred, val}_2\text{)} \wedge \ldots \text{fonc}_n\text{(Nom\_pred, val}_n\text{)}$$

Il existe bien d'autres façons de représenter la même phrase que nous ne développerons pas. Nous voulons simplement insister sur le fait que si la logique fournit un cadre de représentation, elle ne fournit en aucun cas les moyens de choisir quelle est la meilleure de ces techniques : le choix des représentations est laissé au programmeur qui doit prendre grand soin d'utiliser des méthodes cohérentes et de s'interroger sur les avantages et les inconvénients de chacune des représentations, par rapport à son problème.

Un des éléments qui peut expliquer le succès de la logique des prédicats est l'existence du principe de résolution ainsi que des techniques de résolution automatique avec les clauses de Horn. Ceci facilite de façon considérable la programmation d'un SE utilisant le calcul des prédicats comme mode de représentation.

16.2.2 Logique modale

Les logiques modales permettent d'enrichir le langage de la logique classique du premier ordre en introduisant des notions de possibilité/nécessité (logique aléthique) ou de croyance. Ainsi, nous avions développé, dans le chapitre 8, le système S5 dont nous avions montré qu'il permettait de représenter les notions de savoir si l'on

prenait comme interprétation des opérateurs modaux : $\Box \equiv$ SAVOIR et $\Diamond \equiv$ COMPATIBLE AVEC MES CONNAISSANCES.

On peut alors exprimer aisément des propositions comme : **Tout ce que sait Paul, Eric le sait** :

$$\forall x, \Box(\text{Paul})x \rightarrow \Box(\text{Eric})x$$

ou encore : **Si quelqu'un sait qu'il fait beau, alors il fait beau** :

$$\exists x, \Box(x)\text{Faire\_beau} \rightarrow \text{Faire\_beau}$$

On peut, avec cette logique, représenter des connaissances assez complexes. Ainsi le problème des sages et des chapeaux[3] :

> « Trois sages sont dans une pièce, chacun portant sur la tête un chapeau noir ou blanc. Ils savent qu'un chapeau au moins est blanc. Au bout de quelques secondes, le premier sage dit : « Je ne connais pas la couleur de mon chapeau ». Puis le second sage dit : « Je ne connais pas non plus la couleur de mon chapeau ». Alors le troisième sage peut conclure (en utilisant brillamment la logique modale) : « Mon chapeau est blanc ». »

Voyons comment peut se représenter le problème (les variables sont toujours prises dans le sens de la généralité dans les formules suivantes) :

- tout d'abord, écrivons qu'il y a au moins un chapeau blanc :

$$\Box(a)((\text{noir}(x) \wedge \text{noir}(y) \wedge x \neq y) \rightarrow (\text{blanc}(z) \wedge z \neq x \wedge z \neq y))$$

  Cette formule peut se lire : tout le monde sait que si le chapeau de x est noir, que le chapeau de y est noir et que x et y sont des personnes différentes, alors le chapeau de z sera blanc et z sera une personne différente de x et y.
- exprimons maintenant le fait que chaque sage voit le chapeau des deux autres :

$$\Box(a)((\Diamond(x)\text{blanc}(y) \wedge x \neq y) \rightarrow (\Box(x)\text{blanc}(y)))$$

$$\Box(a)((\Diamond(x)\text{noir}(y) \wedge x \neq y) \rightarrow (\Box(x)\text{noir}(y)))$$

  La première formule exprime que tout le monde sait que, s'il est compatible avec les connaissances de x que le chapeau de y soit blanc, et que l'on a $x \neq y$, alors x sait que le chapeau de y est blanc. En effet, x voyant le chapeau de y, s'il est compatible avec ses connaissances que le chapeau est blanc, c'est qu'il l'est. La seconde formule exprime la même propriété pour la couleur noire[4].
- il nous faut maintenant représenter les informations que donnent la réponse du premier sage :

$$\Box(x)(\Box(y)(\Diamond(\text{sage1})\text{noir}(\text{sage1})))$$

---

3 Aussi connu sous le nom de « la tâche sur le front des philosophes ».

4 On peut, à cette occasion, se rappeler la remarque du paragraphe précédent concernant les prédicats unaires et binaires. On aurait pu utiliser un prédicat binaire chapeau et écrire la formule précédente :

$$\forall a, \Box(a)((\Diamond(x)\text{chapeau}(y, c) \wedge x \neq y) \rightarrow (\Box(x)\text{chapeau}(y, c)))$$

où c peut prendre les valeurs noir ou blanc.

Nous pouvons lire cette formule : toute personne x sait que toute personne y sait qu'il est compatible avec les connaissances du « sage1 » qu'il ait un chapeau noir. En effet, le premier sage a été incapable de répondre, donc il sait qu'il est possible qu'il ait un chapeau noir, et chacun sait que les autres le savent[5]. Nous devons, bien entendu, écrire la même formule en remplaçant noir par blanc.

- nous devons prendre en compte la réponse du second sage :

$$\Box(x)\Box(y)(\Diamond(\text{sage2})\text{noir}(\text{sage2})))$$

pour les mêmes raisons que précédemment.

Il est maintenant possible de démontrer la formule $\Box(\text{sage3})\text{blanc}(\text{sage3})$.

La logique modale permet donc de raffiner considérablement le langage de la logique classique. Elle peut être utilisée dès que le problème inclut des notions de croyance, de possibilité, ou de nécessité.

Il est également possible de faire de la résolution automatique en logique modale, bien que ce soit autrement plus difficile qu'en logique classique. Pour une présentation de la résolution en logique non classique, voir (Fariñas del Cerro 1982; Enjalbert and Fariñas del Cerro 1989; Alliot and Herzig 1992; Alliot 1992; Baudinet 1989).

16.2.3 Logique temporelle

Nous avons également présenté une logique temporelle au chapitre 8. À cette occasion, nous avons montré comment l'utiliser pour représenter les connaissances liées à l'exécution d'un programme.

Il existe diverses approches de la logique temporelle. L'une d'entre elles, celle d'Allen (Allen 1981), s'appuie avant tout sur la notion d'intervalle[6]. Elle semble connaître un certain succès dans les systèmes experts à l'heure actuelle, c'est pourquoi nous allons en dire quelques mots.

Allen définit les opérateurs suivants :

- **Pendant**(i_1,i_2) : L'intervalle i_1 est totalement contenu dans i_2.
- **Avant**(i_1,i_2) : L'intervalle i_1 précède i_2. Les deux intervalles ne se chevauchent pas.
- **Acheval**(i_1,i_2) : L'intervalle i_1 commence avant i_2 et les deux intervalles se chevauchent.
- **Joint**(i_1,i_2) : L'intervalle i_1 commence avant i_2 mais ils ne sont séparés par aucun intervalle.
- **Egal**(i_1,i_2) : Les deux intervalles sont égaux.

Les relations sont formalisées par un système d'axiomes dont voici des exemples :

---

5 Il est inutile d'inclure la formule

$$\Box(x)(\Diamond(\text{sage1})\text{noir}(\text{sage1})))$$

car elle découle de celle-ci et de l'axiome T de S5.

6 Un intervalle est une partie convexe de l'espace temporel, souvent assimilé à $\mathbb{R}$. D'autres logiques temporelles s'appuient sur les instants (par exemple) comme élément de base. Dans tous les cas, on peut décrire un intervalle (fermé par exemple) par une paire d'instants et s'intéresser aux positions *numériques* de ces instants ou uniquement aux relations dites « *symboliques* » liant les intervalles (ou les instants) entre eux. C'est cette dernière approche qui est le plus souvent choisie en IA. Les deux mondes, qui ne sont pas si distants qu'il y parait au premier abord, peuvent éventuellement coopérer.

- $\text{Avant}(i_1, i_2) \wedge \text{Avant}(i_2, i_3) \rightarrow \text{Avant}(i_1, i_3)$
- $\text{Joint}(i_1, i_2) \wedge \text{Pendant}(i_2, i_3) \rightarrow \text{Acheval}(i_1, i_3) \vee \text{Pendant}(i_1, i_3) \vee \text{Joint}(i_1, i_3)$

Allen introduit également le prédicat $\text{Satisfait}(p, i)$ qui signifie que la propriété p est satisfaite pendant l'intervalle i. Cela lui permet de formaliser les relations sur les connecteurs classiques en écrivant des relations de la forme :

- $\text{Satisfait}(\text{et}(p, q), i) \leftrightarrow \text{Satisfait}(p, i) \wedge \text{Satisfait}(q, i)$
- $\text{Satisfait}(\text{tout}(x), p) \leftrightarrow \forall x \text{Satisfait}(x, p)$
- ...

Le deuxième élément constitutif de la logique d'Allen est la notion d'*événements*. Il définit ainsi un prédicat :

$$\text{Changepos}(\text{avion}, x, y, e)$$

qui exprime que e est un événement qui consiste dans le passage de l'avion du point x au point y. Il peut alors écrire :

$$\text{Arrive}(e, i)$$

qui signifie que l'événement e arrive pendant l'intervalle i. Il est alors possible d'écrire les conditions d'occurrence d'un événement :

$$\begin{aligned}&\text{Changepos}(\text{objet}, \text{origine}, \text{destination}, e) \wedge \text{Arrive}(e, i) \leftrightarrow \\ &\quad \exists i_1, i_2, \text{Joint}(i_1, i) \wedge \text{Joint}(i, i_2) \wedge \\ &\quad \text{Satisfait}(\text{en}(\text{objet}, \text{origine}), i_1) \wedge \text{Satisfait}(\text{en}(\text{objet}, \text{destination}), i_2)\end{aligned}$$

Allen définit en tout quarante-quatre axiomes pour sa logique. Se pose évidemment le problème de la consistance d'un tel ensemble d'axiomes (Turner semble en douter (Turner 1986b)). D'autre part, il est très difficile de se représenter ce que pourrait être un modèle du système axiomatique d'Allen, ce qui rend difficile toute interprétation sémantique.

16.2.4 Logique multivaluée et logique floue

Nous ne présenterons que rapidement la logique multivaluée (celle de Lukasiewicz) et la logique floue. Disons qu'elles tentent de formaliser des raisonnements du type : « il est très vrai que ... », ou « il est probablement vrai que ... » La logique multivaluée de Lukasiewicz définit ses valeurs de vérité sur l'intervalle de $\mathbb{R}$ $[0, 1]$, et définit les connecteurs logiques par :

- $\neg r = 1 - r$
- $p \wedge q = \min(p, q)$
- $p \vee q = \max(p, q)$
- $p \rightarrow q = \min(1, 1 - p + q)$

Une autre approche (logique floue) est celle de Zadeh (Zadeh 1965; 1974) qui considère plutôt une partition de $[0, 1]$ en un ensemble dénombrable de sous-intervalles, chaque intervalle représentant une valeur de vérité du type : {vrai, faux, pas vrai, pas faux, plus ou moins vrai, plus ou moins faux, plutôt vrai, plutôt faux, ... }. L'inférence en logique floue est radicalement différente de ce qu'elle est en logique classique. Ainsi, une inférence peut être :

$$\frac{\text{a petit , a et b approximativement semblables}}{\text{b est plus ou moins petit}}$$

En fait, la logique floue présente un ensemble de caractéristiques (périphérie des problèmes de complétude et de consistance, caractérisation de la validité purement sémantique, valeurs de vérité subjectives) qui la rend très différente de la logique classique. Certains auteurs vont même jusqu'à estimer que l'on peut difficilement dire de la logique floue qu'il s'agit d'une logique. En fait, la seule justification de la logique floue est son intérêt comme outil opératoire : elle semble, en effet, rencontrer d'intéressantes applications dans le domaine des systèmes experts. Il existe des formalisations de certaines logiques floues permettant de réaliser des inférences de façon automatique. Pour plus de précisions on peut se reporter à (Turner 1986b; Dubois and Prade 1988; Kaufman 1973).

16.2.5 Logique des défauts

La logique des défauts tente de modéliser un type de raisonnement, que nous pratiquons tous, qui consiste à réaliser des inférences en fonction de règles que nous supposons générales alors qu'elles admettent parfois des exceptions. Si, par exemple, je dis *l'*EOLE *est un avion.* J'en déduirais *l'*EOLE *vole.* Pourtant, comme l'EOLE est l'avion de Clément Ader conservé au musée de l'Air, je me trompe. La règle d'inférence que j'applique est :

$$\frac{\text{Avion}(x) : M\text{Vole}(x)}{\text{Vole}(x)}$$

que l'on peut lire : « si x est un avion et qu'il est consistant avec mes connaissances présentes que x vole, alors x vole. » Ce type de raisonnement est appelé *raisonnement par défaut.* La logique des défauts appartient à une classe plus vaste de logiques, les logiques non-monotones ainsi appelées car les raisonnements qu'elles modélisent sont *révisables* et l'ensemble des formules déductibles ne croit pas (au sens de l'inclusion) de façon monotone lorsque l'on ajoute une formule dans les hypothèses. En effet, si j'ai induit EOLE vole par une règle par défaut et que par la suite mon état de connaissance évolue et m'indique que EOLE est bloqué au musée de l'air pour toujours, je devrais modifier mon inférence initiale et supprimer le fait inféré et les raisonnements qui en découlent.

De façon plus formelle, un *défaut* ou *règle de défaut* se représente par :

$$\frac{\alpha : M\beta_1, \ldots, M\beta_n}{\gamma}$$

que l'on peut lire : « si α est vérifié et que les β sont consistants dans l'état actuel de mes connaissances, alors γ est vérifié. » α est le pré-requis, les β sont les justifications et γ est le conséquent. Une théorie des défauts est composée d'un ensemble F de formules et d'un ensemble D de défauts[7].

Il est intéressant, en logique des défauts, de considérer les extensions de la théorie (D, F). Il s'agit des ensembles de croyances que l'on peut inférer à partir de (D, F). Pour reprendre un exemple classique (Froidevaux 1986), considérons les assertions suivantes :

---

7 Nous ne tentons pas de définir absolument correctement la théorie des défauts. En particulier, nous n'introduisons pas les notions de défaut ouvert ou fermé, et de théorie ouverte ou fermée. On peut se rapporter à (Thaysse and others 1988) ou (Reiter 1987) pour plus de précisions.

- les mollusques sont des coquillages ;
- les céphalopodes sont des mollusques mais ne sont pas des coquillages (exception) ;
- les nautiles sont des céphalopodes et sont des coquillages (exception à l'exception).

On peut transformer ces affirmations en la théorie des défauts (D, F) suivante :

Défaut 1 Si x est un mollusque, et s'il est consistant avec mes connaissances que x est un coquillage et n'est pas un céphalopode, alors x est un coquillage :

$$\frac{\text{Mollusque}(x) : M(\text{Coquillage}(x) \wedge \neg\text{Céphalopode}(x))}{\text{Coquillage}(x)}$$

Défaut 2 Si x est un céphalopode et s'il est consistant avec mes connaissances que x n'est pas un coquillage et n'est pas un nautile, alors x n'est pas un coquillage :

$$\frac{\text{Céphalopode}(x) : M(\neg\text{Coquillage}(x) \wedge \neg\text{Nautile}(x))}{\neg\text{Coquillage}(x)}$$

Formule 1 Les nautiles sont des céphalopodes :

$$\forall x, \text{Nautile}(x) \rightarrow \text{Céphalopode}(x)$$

Formule 2 Les céphalopodes sont des mollusques :

$$\forall x, \text{Céphalopode}(x) \rightarrow \text{Mollusque}(x)$$

Formule 3 Les nautiles sont des coquillages :

$$\forall x, \text{Nautile}(x) \rightarrow \text{Coquillage}(x)$$

Si x est un nautile, les formules 1, 2 et 3 nous permettent d'inférer que x est un céphalopode, un mollusque et un coquillage. Cette inférence est l'extension de la théorie formée par $(D, F) \cup \{x \text{ est un nautile}\}$. Si x est un céphalopode et si x n'est pas un nautile, alors on infère de la formule 2 que x est un mollusque et du défaut 2, que x n'est pas un coquillage. Il s'agit là de l'extension de la théorie $(D, F) \cup \{x \text{ est un céphalopode} \wedge x \text{ n'est pas un nautile}\}$.

Il existe deux sous-catégories importantes de logique des défauts : les théories *normales* et les théories *semi-normales*. Dans une théorie normale, tous les défauts sont de la forme :

$$\frac{\alpha : M\beta}{\beta}$$

Les théories normales présentent plusieurs avantages : elles admettent toujours une extension, et si on augmente l'ensemble des défauts, la nouvelle théorie normale admet une extension qui inclut l'extension de la théorie d'origine (*semi-monotonie*). Malheureusement, les théories normales présentent certains inconvénients liés à la transitivité des règles de défaut. Considérons les deux règles normales de défaut suivantes[8] :

8 Instructeur CA : Instructeur Circulation Aérienne, Contrôleur : Contrôleur Aérien, CCR : Centre de Contrôle en Route.

– les instructeurs CA sont des contrôleurs :

$$\frac{\text{InstructeurCA}(x) : M\text{Contrôleur}(x)}{\text{Contrôleur}(x)}$$

– les contrôleurs travaillent dans des CCR :

$$\frac{\text{Contrôleur}(x) : M\text{TravailleCCR}(x)}{\text{TravailleCCR}(x)}$$

On constate que l'on peut inférer que les instructeurs CA travaillent dans des CCR, ce qui est inexact. On peut résoudre partiellement le problème en introduisant un troisième défaut normal qui indiquerait qu'un instructeur CA ne travaille pas dans un CCR. Cependant, cette technique nous donne une théorie qui admet deux extensions pour un instructeur CA particulier, une qui affirme qu'il travaille dans un CCR et l'autre qu'il n'y travaille pas. Or nous souhaiterions favoriser la seconde extension. Il faut alors modifier notre théorie normale en une théorie *semi-normale* dont les défauts seraient :

1. les instructeurs CA sont des contrôleurs :

$$\frac{\text{InstructeurCA}(x) : M\text{Contrôleur}(x)}{\text{Contrôleur}(x)}$$

2. les instructeurs CA ne travaillent pas dans des CCR :

$$\frac{\text{InstructeurCA}(x) : M\neg\text{TravailleCCR}(x)}{\neg\text{TravailleCCR}(x)}$$

3. les contrôleurs travaillent dans des CCR s'ils ne sont pas instructeurs CA :

$$\frac{\text{Contrôleur}(x) : M(\text{TravailleCCR}(x) \wedge \neg\text{InstructeurCA}(x))}{\text{TravailleCCR}(x)}$$

Cette théorie possède la bonne extension pour un instructeur CA x : {InstructeurCA(x), Contrôleur(x), ¬TravailleCCR(x) }. Le défaut 3 est un défaut *semi-normal.* Les défauts semi-normaux sont de la forme :

$$\frac{\alpha : M(\beta \wedge \neg\gamma)}{\beta}$$

Les défauts semi-normaux sont contrôlés par la condition γ supplémentaire. Malheureusement les théories semi-normales perdent la propriété de semi-monotonie, et ne possèdent pas forcément d'extension.

16.2.6 Réseaux sémantiques

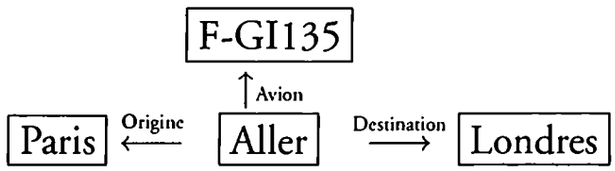
Les réseaux sémantiques constituent, très grossièrement, une façon de représenter graphiquement les relations de la logique du premier ordre. Reprenons l'exemple :

Aller(F-GI135, Paris, Londres)

Nous avons vu que l'on peut le décomposer en :

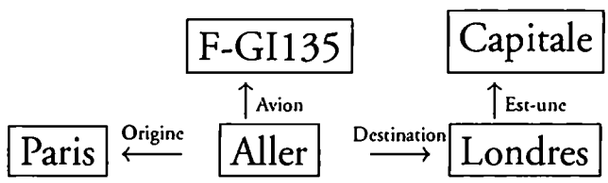
$$\text{Avion}(\text{Aller}, \text{F-GI135}) \wedge \text{Origine}(\text{Aller}, \text{Paris}) \wedge \text{Destination}(\text{Aller}, \text{Londres})$$

Cette formule de la logique du premier ordre peut se représenter sous la forme :

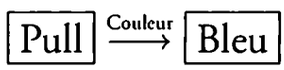


Nous voyons comment des prédicats binaires peuvent ainsi se représenter sous forme de réseau. L'intérêt de la représentation sous forme de réseau sémantique est que les informations sont groupées autour d'*instances*, donnant de l'information une vision plus structurée. Nous pouvons aisément enrichir ce réseau en ajoutant par exemple que Londres appartient à l'ensemble des capitales :

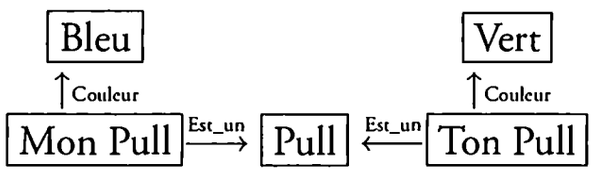
$$\text{Est\_une}(\text{Capitale}, \text{Londres})$$



L'intérêt de la notion d'instance apparaît clairement. Si nous voulons représenter la phrase : « le pull est bleu », nous obtenons le réseau :



Mais supposons que nous souhaitions ajouter la phrase : « ton pull est vert ». Nous allons devoir utiliser le prédicat *Est_un* :



Les réseaux sémantiques permettent de répondre à des questions comme « quel est le lien entre Londres et Paris ? » : « F-GI135 va de Londres à Paris. »

Un problème lié à l'utilisation des réseaux sémantiques est la représentation des quantificateurs. Ainsi, « toutes les capitales sont des villes » pourra se représenter par : Est-Une-Sorte-de(Capitale, Ville) dénotant une relation d'inclusion. Différentes techniques ont été développées dont celle des réseaux partitionnés (Hendrix 1977).

Les réseaux sémantiques ont été utilisés, en particulier, pour la compréhension des langues naturelles. Ils peuvent également servir de représentation intermédiaire entre une connaissance non formalisée et un langage de programmation.

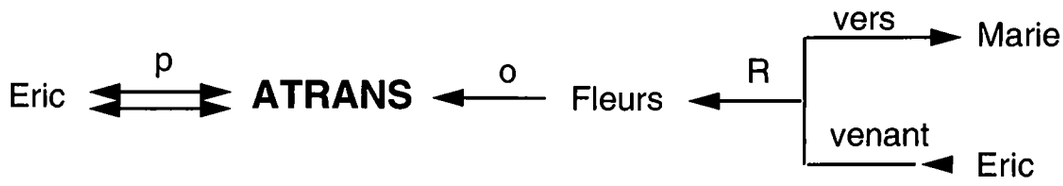


Figure 16.1 – Eric a donné des fleurs à Marie

16.2.7 Dépendance conceptuelle

La dépendance conceptuelle (DC) a pour but de représenter le sens des phrases (en langage naturel) de façon à faciliter la déduction d'inférence et à rester indépendant de la langue dans laquelle la phrase a été énoncée. La représentation en DC d'une phrase n'est pas construite à partir des mots composant la phrase mais à partir de primitives conceptuelles (d'où le nom de la théorie).

Voyons cela sur un exemple. Soit la phrase : « Eric a donné des fleurs à Marie ». La représentation en DC de cette phrase se trouve sur la figure 16.1.

Nous pouvons rapidement décrire les primitives utilisées :
- une flèche indique une relation de dépendance ;
- une double flèche indique un lien dans les deux sens entre un acteur et une action ;
- **ATRANS** est une des primitives conceptuelles. Elle indique un transfert d'une relation abstraite (donner) ;
- **p** indique le temps du verbe (passé) ;
- **o** indique qu'il s'agit d'un objet ;
- **R** indique qu'il s'agit du receveur.

La DC comprend tout un ensemble de primitives conceptuelles qui sont :
- **ATRANS** : transfert d'une relation abstraite ;
- **ATTEND** : focalisation de l'attention sur un signal ;
- **EXPEL** : expulsion de quelque chose du corps ;
- **GRASP** : prise d'un objet par un acteur ;
- **INGEST** : absorption d'un objet ;
- **MBUILD** : construction d'une information ;
- **MOVE** : mouvement d'une partie du corps ;
- **MTRANS** : transfert d'information ;
- **PROPEL** : application d'une force sur un objet ;
- **PTRANS** : transfert de la position d'un objet ;
- **SPEAK** : production de sons.

Schank sépare quatre catégories conceptuelles primitives :
- **ACT** : actions ;
- **AA** : modificateur d'action ;
- **PP** : objets (producteurs d'image) ;
- **PA** : modificateurs de PP.

Il est alors possible, en utilisant les connecteurs, de définir quatorze relations de dépendance de base entre les quatre catégories conceptuelles.

L'utilisation des réseaux de dépendance conceptuelle est resté limitée, à notre connaissance, au domaine du langage naturel. Pour une présentation précise, on peut lire (Schank 1975).

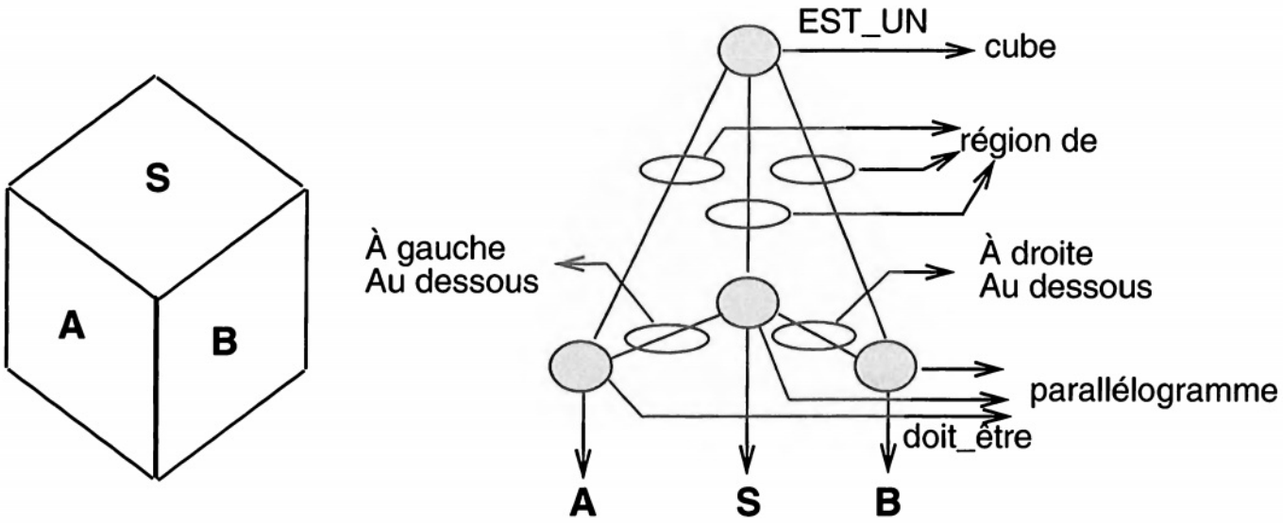


Figure 16.2 – Représentation d'une vue d'un cube par un *frame*

16.2.8 Frames

Les *frames* sont des descriptions qui scindent la représentation d'une classe d'objet en plusieurs composantes distinctes, ou *cases* ou *attributs* (slots en anglais). Pour qu'un objet donné appartienne à la classe et que l'on puisse *instancier un frame*, il faut que l'on puisse remplir certaines des cases de la description par des paramètres caractérisant cet objet (preuve partielle). Si l'on ne parvient pas à trouver de *frames* en concordance avec l'objet, il faut sélectionner la plus proche et tenter d'expliquer l'échec de la concordance. Certaines des cases d'un *frame* peuvent contenir des valeurs par défaut. Ceci permet de réaliser des inférences par défaut de façon simple et rapide. Nous voyons un exemple de *frame* sur la figure 16.2, inspiré directement de (Minsky 1975). Les arcs *doit_être* ou *région_de* sont des contraintes qui doivent être satisfaites pour que la concordance soit reconnue. L'arc *EST_UN* donne une information sur l'objet quand la concordance est reconnue.

Les *frames* sont des structures qui peuvent être utilisées pour représenter différents types d'objets. Comme les réseaux sémantiques, ils peuvent servir d'interface entre des représentations non formelles et des langages de programmation. Le développement des techniques de représentation des connaissances par *frames* est (partiellement) à l'origine d'un nouveau type de langages, les langages *à objets*. Ces langages ne donnent pas la primauté à la structure du contrôle du programme (langage algorithmique classique) ni au découpage fonctionnel (programmation fonctionnelle) ou à la structure logique (programmation logique), mais à l'organisation des données. Un exemple de langage-objet directement issu des principes de représentation par *frames* est SMALLTALK.

16.2.9 Scripts

Les scripts sont des représentations figées de séquences d'événements qui décrivent une scène particulière (d'où leur nom). Les scripts utilisent fréquemment les primitives de la DC pour représenter les relations entre actions.

L'utilisation des scripts permet, à partir d'événements fractionnaires, de reconstituer un grand nombre d'informations sous-entendues (par défaut). Ainsi, si on me dit : « Paul

va au restaurant », j'en déduis en général que Paul a faim et qu'il a de l'argent. Un ordinateur qui utilise un script en est également capable. Il est également capable d'inférer toutes les actions que Paul va faire en allant au restaurant. Si on lui dit : « Paul est parti sans payer », il en déduira qu'il n'y avait pas la nourriture appropriée et que Paul n'a pas mangé.

Une grande faiblesse des scripts est leur inaptitude à réagir face à une séquence imprévue. Ainsi, soit la phrase : « Victoria est allée au restaurant. Elle a trouvé un cafard dans sa salade, s'est plainte au chef et est partie. » et voilà notre ordinateur perdu. Impossible pour lui de répondre à la question : « Victoria a-t-elle payée? » En revanche, le script peut détecter l'événement anormal qui interrompt la séquence « a trouvé un cafard dans la salade, s'est plainte au chef ».

Il existe nombre de situations où les scripts peuvent être utiles : les accidents de voiture, une interrogation écrite..., en fait, toute séquence stéréotypée d'événements. Il ne faut cependant pas en attendre plus que ce qu'ils peuvent donner : un script ne peut s'écarter de ce qu'il sait traiter et les événements exceptionnels doivent faire l'objet d'un traitement séparé.

16.3 SE à base de règles

Un système expert (par règles) s'articule autour de trois éléments principaux :

La base de faits : elle contient l'ensemble des faits relatifs au problème considéré. C'est là que sont stockées les données symboliques et numériques. Par exemple, en logique propositionnelle, il s'agit des affirmations concernant la valeur de vérité des propositions (p est vrai, q est faux...).

La base de règles : elle contient l'ensemble des règles de production et des connaissances heuristiques déterminant la résolution du problème. En logique propositionnelle, il s'agit des hypothèses de la forme $p \rightarrow q$, $(p \vee q) \leftrightarrow r \ldots$

Le moteur d'inférence : il réalise le fonctionnement créatif du système en sélectionnant règles et faits pour générer de nouvelles règles et de nouveaux faits. En logique propositionnelle, il s'agit des règles d'inférence pour le chaînage avant : $\frac{\vdash p, \vdash p \rightarrow q}{\vdash q}$ et des règles de déduction pour le chaînage arrière.

On inclut parfois deux éléments supplémentaires dans les systèmes experts :

Les méta-règles : il s'agit de règles destinées à guider le moteur d'inférence quant à la stratégie de résolution ou à la maintenance des bases de règles et de faits. Ainsi, lorsque nous avons introduit en logique propositionnelle l'algorithme de Davis et Putnam comme méthode de résolution, nous avons signalé qu'il était judicieux d'utiliser un mode particulier de sélection des clauses pour que cet algorithme soit efficace : c'est un exemple typique de méta-règle. On peut bien entendu avoir des méta-(méta-règles), des méta-(méta-(...))règles.

Les agendas : il s'agit d'un moyen de gérer la notion de plans et de planification. L'agenda est une liste de tâches que le système peut effectuer avec les « raisons » qui poussent le système à penser qu'une tâche donnée est « intéressante ». Ainsi le système peut choisir la tâche la plus intéressante pour poursuivre la résolution. En

résolution logique, nous pourrions chaque fois que nous générons une nouvelle clause, noter les raisons qui la rendent intéressante à sélectionner.

La première étape pour la réalisation d'un système expert est le choix de la technique de représentation des connaissances (aussi bien faits que règles) que nous allons utiliser. Nous avons décrit dans la section précédente différentes techniques de représentation des connaissances et nous n'y revenons pas. La seconde étape consiste à choisir un mécanisme d'inférence adapté au problème.

16.3.1 Les moteurs d'inférence

Le moteur d'inférence est la partie créative du système. À partir de règles et de faits, il génère de nouveaux faits afin de réaliser la résolution effective du problème.

La majorité des moteurs d'inférence utilise un cycle à trois temps (MATCH, CONFLICT RESOLUTION, ACT)[9] :

MATCH (Sélection) : l'étape MATCH consiste à rechercher l'ensemble des règles d'inférence qui peuvent s'appliquer à un instant donné à la base de faits, et à noter les faits à utiliser. Ainsi, en résolution avec des clauses de Horn, si je dois résoudre la question p, le moteur va sélectionner toutes les clauses dont la tête est p.

CONFLICT RESOLUTION (Résolution de conflit) : le moteur va choisir parmi toutes les règles qu'il peut utiliser, celle (ou celles) qu'il va exécuter. Ce choix est fait à l'aide de méta-règles. Un exemple typique consiste à choisir la première règle disponible.

ACT (Déclenchement) : après avoir sélectionné la règle, le moteur d'inférence l'exécute et exécute toutes les actions associées (remise à jour de la base...).

Les caractéristiques principales d'un moteur d'inférence sont liées aux techniques de résolution qu'il implante.

Chaînage : un moteur d'inférence peut fonctionner en *chaînage avant* ou en *chaînage arrière*. Un moteur d'inférence fonctionnant en chaînage avant tente, à partir de la base de faits et des règles, de générer une réponse : une démonstration classique (déductive) est un exemple de fonctionnement en chaînage avant.

Un moteur d'inférence à chaînage arrière tente d'invalider la question en montrant l'inconsistance de la nouvelle base de faits comprenant la négation de la question. Le mécanisme de résolution avec des clauses de Horn est un exemple typique de fonctionnement en chaînage arrière.

Il est également possible d'utiliser le *chaînage mixte* qui tente de concilier les deux méthodes. L'utilisation du chaînage mixte demande la mise en place d'heuristiques particulières capables de décider face à une situation donnée quel est le type de stratégie le plus adapté pour poursuivre la résolution.

Régime de contrôle : le régime de contrôle peut être *irrévocable* ou *à tentative*. Si le système est à régime de contrôle irrévocable, le moteur d'inférence s'arrêtera dès que l'ensemble des règles sélectionnables sera vide. En revanche, dans un système à tentative, le moteur réalisera un *retour-arrière* (backtrack), et remettra en cause les règles déclenchées précédemment.

---

9 Certains auteurs séparent la phase 1 (MATCH) (Sélection) en deux phases : sélection et filtrage.

Monotonie : le moteur d'inférence peut être *monotone* ou *non-monotone*. En régime monotone, le moteur ne fait qu'ajouter des faits à la base de faits. En régime non-monotone, le moteur peut, en cas de retour-arrière par exemple, retrancher de la base des faits qui pourraient se révéler contradictoires.

16.3.2 Systèmes de maintien de cohérence

Lors de l'exploration d'un espace de recherche (ou espace d'états), un *résolveur de problème* (en particulier, un système expert) doit, dans chacun des états, effectuer des choix pour atteindre son but. Selon la nature du problème et du résolveur, il peut s'agir d'ensemble de règles à déclencher, d'actions à exécuter, etc. Ces décisions peuvent s'avérer infructueuses, aboutissant à un état d'échec dans lequel il est nécessaire de remettre en cause certains choix. Comme le dit (Palmade 1992) :

> « Lorsque le processus de recherche suit un chemin direct, unique, sans étape inutile, depuis l'énoncé jusqu'à la solution (ou à la conclusion qu'il n'en existe pas), on se trouve plus généralement dans le domaine, disons pour résumer, du calcul numérique. Mais lorsque l'univers se fait labyrinthe et que la recherche devient errance, alors commence l'Intelligence Artificielle, ses algorithmes hésitants et, malheureusement, la complexité exponentielle. »

La méthode la plus habituelle consiste à remettre en cause le dernier choix effectué avant la découverte d'une contradiction (backtrack chronologique), puis de reprendre l'exploration. La méthode est simple mais extrêmement inefficace. Parmi les différentes approches destinées à remédier au caractère « pathologique » du backtrack, nous nous proposons de présenter rapidement les « systèmes de maintien de cohérence » (Reason or Truth Maintenance Systems) qui sont destinés à collaborer avec un système d'exploration afin de lui éviter certaines redondances, via la fourniture d'un service, plus ou moins sophistiqué et complet, de signalisation du « labyrinthe » exploré : impasses, distances, directions à prendre pour atteindre tel état, etc. Depuis, un certain nombre d'autres utilisations ont été trouvées aux TMS : raisonnement abductif (explication), raisonnement non-monotone...

Deux approches essentielles sont généralement retenues : celle des JTMS (Doyle 1979) (Justification based TMS) et celle des ATMS (de Kleer 1986a; 1986b; 1986c; Cayrol *et al.* 1993) (Assumption based TMS), plus récente. Nous nous focaliserons sur cette dernière.

Buts de l'ATMS

Un ATMS constitue un sous-système d'un système de résolution comprenant aussi un « résolveur de problème »[10] (cf. figure 16.3). Dans la pratique, le résolveur de problème fournit à l'ATMS, toutes les inférences qu'il peut réaliser en distinguant, parmi les données fournies, celles qui correspondent à des choix possibles (appelées hypothèses). Dans la pratique, ce sont des formules de la logique des propositions qui sont manipulées.

---

10 Dans la pratique, bon nombre de générateurs de systèmes experts incluent les fonctionnalités d'un TMS. Citons par exemple les systèmes ART et KEE.

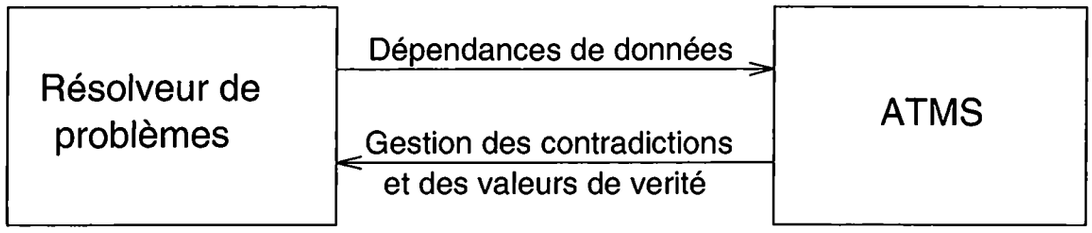


Figure 16.3 – Liens entre ATMS et résolveur de problèmes

L'ATMS explore alors toutes les combinaisons consistantes d'hypothèses[11] et calcule pour chaque donnée tous les ensembles d'hypothèses (de choix) qui permettent d'aboutir à cette donnée.

Principes de l'ATMS

Les données traitées par un ATMS sont formées de :
- un ensemble $\mathcal{L}$ de littéraux, ou variables propositionnelles ;
- un ensemble $\mathcal{K}$ de clauses construites à partir de ces littéraux. Elles sont appelées *justifications*. Chaque clause correspond à une inférence possible du « résolveur de problème » ;
- un ensemble $\mathcal{H}$ d'hypothèses, $\mathcal{H} \subset \mathcal{L}$. Il correspond aux choix possibles du « résolveur de problème ». Ce sont sur ces variables que l'on pourra éventuellement faire du raisonnement hypothétique (que se passe-t-il si x ?) et abductif (pourquoi x ?).

On appelle *environnement* tout sous-ensemble de $\mathcal{H}$. On dira qu'un environnement $\mathcal{E}$ est consistant si $\mathcal{E} \cup \mathcal{K}$ est satisfiable, qu'un littéral $l \in \mathcal{L}$ est valide dans un environnement $\mathcal{E}$ si l découle logiquement ($\models$) de $\mathcal{E} \cup \mathcal{K}$. On appelle *contexte d'un environnement*, l'ensemble des données valides dans cet environnement.

Les deux rôles de l'ATMS sont :

1. le calcul des *nogoods* minimaux : on appelle *nogood* tout environnement inconsistant. L'ATMS doit maintenir l'ensemble $\mathcal{N}$ des nogoods minimaux (aucun élément de $\mathcal{N}$ n'est un sur-ensemble propre d'un autre élément de $\mathcal{N}$). Il faut noter qu'un ensemble de choix inconsistant est alors très simple à détecter : il contient un des nogoods maintenus par l'ATMS. La tâche de calcul des nogoods minimaux étant très coûteuse, quelques implantations se limitent à la maintenance d'une partie de ces ensembles seulement.
2. le calcul des *labels* : étant donné un littéral $a \in \mathcal{L}$, on appelle label de a l'ensemble des environnements $\{\mathcal{E}_i\}$ vérifiant les propriétés suivantes :

   (sain) a est valide dans tous les $\mathcal{E}_i$;

   (consistant) tous les $\mathcal{E}_i$ sont consistants ;

   (complet) tout environnement consistant dans lequel a est valide est un sur-ensemble d'un $\mathcal{E}_i$;

   (minimal) il n'existe pas de $\mathcal{E}_i$ qui soit un sous-ensemble propre d'un autre $\mathcal{E}_i$.

11 À la différence des JTMS, il n'y a pas d'état courant de l'ensemble des hypothèses lorsqu'on utilise un ATMS et jamais de « retour-arrière » puisque les ensembles d'hypothèses inconsistants sont évités. On parle de TMS mono-contexte (JTMS) et multi-contexte (ATMS).

Les labels permettent de savoir dans quels environnements une donnée est vraie. Ils permettent aussi le calcul du contexte d'un environnement.

Dans la pratique, on considère souvent les nogoods comme les éléments du label d'un littéral fictif, noté « $\perp$ » et dénotant l'inconsistance (Ce « label » ne vérifie évidemment pas la propriété de consistance).

L'ensemble des labels/nogoods permet de répondre aux questions suivantes de la part du résolveur :

- est-il consistant de supposer $\{a, b, c \ldots\}$? (nogoods). Ceci permet d'éviter tout retour-arrière, puisque le résolveur n'effectuera jamais un choix qui ne peut pas mener à un état consistant ;
- que dois-je supposer pour que l soit vrai ? Ou quelles sont les explications minimales de l ? (labels permettant un raisonnement abductif) ;
- que se passe-t-il si je suppose $\{a, b, c \ldots\}$? (contextes, permettant un raisonnement hypothétique).

Les algorithmes utilisés pour calculer ces ensembles reposent sur un certain nombre de règles d'inférence. Certaines de ces règles (celles concernant le calcul des « nogoods ») peuvent être rapprochées des techniques d'établissement de consistance locale définies dans le cadre des CSP (de Kleer 1989).

La tâche de l'ATMS est très lourde. Dans sa version initiale, qui se limite au traitement de clauses de Horn, l'ATMS définit déjà un problème NP-complet. On a pu montrer (Provan 1990) que dans un cadre plus large (ATMS traitant des clauses quelconques), les problèmes traités sont extrêmement complexes, puisqu'ils sont NP-durs, ou plus précisément Σ_2^p-complets[12]. Aussi la majorité des systèmes se limitent-ils à la recherche de certains « nogoods » seulement et la propriété de consistance des labels est, de ce fait, affaiblie.

16.3.3 Langages d'implantation

La plupart des systèmes experts à base de règles ont été écrits pendant de nombreuses années dans un langage généraliste de l'IA : LISP. Plus récemment, PROLOG est devenu extrêmement populaire. PROLOG présente la particularité d'avoir un moteur d'inférence intégré capable de travailler sur des clauses de Horn suivant les mécanismes de résolution décrits dans le chapitre 7. Il existe également des langages plus spécialisés dans l'écriture de SE. Ainsi, OPS5, SNARK, CLIPS (et bien d'autres) sont des langages dédiés à l'écriture de SE.

16.4 Exemples de réalisation

Il est impossible de donner une liste exhaustive des systèmes experts écrits. Nous nous contentons de citer les plus célèbres :

---

12 Autrement dit, $\mathsf{NP}^{\mathsf{NP}}$-complets *i.e.*, le problème est parmi les plus durs des problèmes qui, si l'on considère que la résolution d'un problème NP-complet s'effectue en un temps constant (c'est une transition d'une « super machine » de Turing, appelée machine de Turing à oracle NP), restent des problèmes qui se traitent en un temps polynomial via une « super-machine » de Turing *non déterministe*... Pour plus de détails sur la classe des algorithmes Σ_k^p, se reporter à (Garey and Johnson 1979), page 162 ou à (Johnson 1990).

DENDRAL : DENDRAL est l'ancêtre des systèmes experts. Il a été développé à partir de 1964 à l'université de Stanford. Il utilise un moteur d'inférence à chaînage avant monotone par tentative (en fait, il s'agit d'une technique proche de l'énumération). Le but de DENDRAL est de donner la formule développée d'un composé chimique à partir de sa formule brute et de données spectroscopiques. DENDRAL est utilisé régulièrement et on peut considérer que les résultats qu'il donne sont satisfaisants. Il est intéressant de savoir, qu'à l'origine, DENDRAL était écrit dans un langage procédural traditionnel.

META-DENDRAL : META-DENDRAL a pour but d'inférer automatiquement des règles qui serviront par la suite à DENDRAL.

MYCIN : MYCIN fait également partie des systèmes experts devenus mythiques. Développé vers 1974, il utilise une logique « empirique » de type probabiliste pour la représentation des connaissances. Son but est de fournir une aide aux diagnostics et aux traitements des infections bactériennes. MYCIN est effectivement utilisé et donne des résultats satisfaisants. Il est à l'origine de EMYCIN, moteur généraliste de SE.

RITA : RITA a été développé par la RAND corporation vers 1976. Il s'agit d'un système d'aide à la conception. RITA est également un des ancêtres des systèmes experts modernes.

SHRUDLU : il est difficile de savoir si SHRUDLU est un système expert à proprement parler. SHRUDLU simule un petit robot manipulateur de cubes avec lequel l'utilisateur peut dialoguer en langage naturel, via un clavier. Développé par Winograd, l'intérêt de SHRUDLU repose, avant tout, sur son interface utilisateur.

HEARSAY-III : HEARSAY-III est un système expert pour la compréhension de la parole *continue*. HEARSAY-III a été développé à partir du système HEARSAY de Carnegie Mellon, qui utilise une approche dite *mémoire auxiliaire*[13]. Le système est un ensemble de sources de connaissances (Knowledge Sources, KS en anglais) indépendantes et d'une mémoire auxiliaire accessible à toutes les KS. Une description plus précise de HEARSAY-III se trouve dans (Rich 1987).

SU/X : SU/X a été développé aux alentours de 1978. Il tente d'identifier des sources radio-émettrices dans l'espace. SU/X utilise les techniques TMS (Truth Maintenance System).

R1/XCON : système expert conçu par Digital Equipment Corporation pour définir des systèmes complets adaptés aux besoins d'un client. XCON a été écrit en OPS. Il comprend plusieurs milliers de règles et est effectivement utilisé.

DART : système expert développé chez IBM pour les pannes de matériel et de systèmes.

ARGOSL : système général développé à Toulouse par M.Cayrol, H. Farreny et B. Fade qui simule le fonctionnement d'un robot.

AM : AM (Lenat 1976) est un système expert qui a pour but de découvrir des concepts et des théorèmes mathématiques. AM utilise une représentation des connaissances par frames. Le moteur d'inférence est à chaînage avant ; il est monotone et à tentative[14].

---

13 Technique appelée *blackboard* en anglais.

14 AM est un des programmes les plus célèbres, mais aussi un des plus critiqués de l'IA (voir (Ritchie and Hanna 1984)). Nous en reparlerons brièvement dans le chapitre 23.

16.5 Critique des systèmes experts

En 1982, Randall Davis publiait un article (Davis 1982b), devenu un classique, dans lequel il faisait le point sur les systèmes experts (qui, à l'époque, étaient pour la plupart des systèmes à base de règles tels que nous venons de les décrire)[15].

L'étude des problèmes de la représentation de l'expertise humaine amène, d'après Davis, les enseignements suivants :

- le savoir est la base de l'ensemble du système ;
- le savoir est souvent inexact, voire incomplet ;
- le savoir est souvent mal spécifié ;
- un amateur devient un expert progressivement.

Donc, il est clair que les systèmes experts doivent être flexibles et transparents, car une grande partie de la vie d'un système expert sera consacrée à de permanents changements et de fréquentes remises à jour et améliorations. Davis estime qu'un système expert connaît un cycle de développement en sept étapes et que sa croissance se poursuivra pendant chacune de ses étapes :

1. conception du système ;
2. développement du système ;
3. évaluation formelle de performances ;
4. évaluation formelle d'acceptabilité ;
5. utilisation dans un environnement-prototype ;
6. développement de plans de maintenance ;
7. mise sur le marché.

À propos de R1/XCON[16] Davis dit : « Une croissance importante et continue du système est un processus normal qui continuera pendant toute sa vie. » reprenant en cela les paroles de Drew McDermott : « Le savoir inclus dans R1 grossit au moins autant pendant la dernière étape de son développement que pendant toute autre étape de son cycle de vie. »

Enfin notait Davis, « rares sont les systèmes experts qui dépassent l'étape (2) de leur évolution. La plupart des systèmes experts sont conçus pour tester des idées sur la représentation des connaissances, le contrôle, l'acquisition... Et quand les concepteurs disent que le système *fonctionne*, ils veulent généralement dire que *l'idée fonctionne.* »

Davis dégageait de cet ensemble de caractéristiques les principes d'architecture suivants. Pour écrire un « bon » système expert :

- il faut séparer la base de connaissances du moteur d'inférences ;
- il faut utiliser des représentations les plus uniforme possible ;

---

15 Les critiques de Davis sont d'autant plus intéressantes qu'elles viennent d'un individu issu du sérail de l'IA au MIT. Si, sur le fond épistémologique, elles rejoignent fortement celles que faisait Dreyfus dix ans plus tôt, elles furent mieux accueillies car elles ne prenaient pas le même tour polémique. Il n'en reste pas moins qu'il s'agit d'une remise en cause, que l'on pourrait presque qualifier de fondamentale, du concept de système expert, et surtout de la possibilité de construire des systèmes experts efficaces ailleurs que dans des domaines où la base de connaissances est fortement causale, et clairement modélisable.

16 En 1982, R1 était un exemple prometteur de système expert réussi et c'est tout à l'honneur de Davis d'avoir noté ce qui allait causer bien des tourments à l'équipe chargée de sa maintenance : la non-maintenabilité de la base de règles, du fait de sa trop grande complexité.

- il faut garder le moteur d'inférences le plus simple possible ;
- il faut utiliser la redondance de l'information.

Malheureusement dit Davis, même en respectant ces principes, ou parfois parce que nous les respectons, nous sommes en face de problèmes bien difficiles à résoudre.

- le domaine d'expertise est restreint ;
- le comportement du système expert est fragile aux limites de l'expertise. Contrairement à un véritable expert, il n'y a pas dégradation progressive, mais effondrement brutal ;
- un seul expert sert de base de référence absolue pour la mise au point du programme ;
- le langage de programmation est trop pauvre et trop strict comparé au langage naturel dans lequel l'expert exprime ses connaissances.
- Il est extrêmement difficile d'utiliser une représentation uniforme comme le demande la règle (4). « L'expérience de MACSYMA montre que seule une multiplicité de représentations permet une implantation efficace d'une grande variété d'algorithmes. » Ce phénomène est, hélas, plus une généralité qu'une exception. En d'autres termes *« il est tellement coûteux d'utiliser la mauvaise représentation qu'il vaut mieux se donner le mal d'utiliser une représentation spécialisée, quitte à en payer le prix »*[17] ;
- les explications que sont capables de fournir les systèmes sont limitées. Ceci est principalement dû au système de représentation par règle de la connaissance. (Si A,B,C alors D). Si l'on demande au système « Pourquoi D ? », il répondra « parce qu'A, B et C ». Ce type de réponse peut être satisfaisant dans des univers où la connaissance est essentiellement empirique (MYCIN en est un bon exemple) mais peu acceptable si nous possédons une connaissance plus profonde du domaine considéré ;
- l'acquisition des connaissances reste difficile, pour le même type de raison. Afin de pouvoir apprendre, il faut pouvoir comprendre. Il est donc nécessaire que le modèle de représentation de la connaissance soit plus riche que de simples règles de type « Si ... alors ... ».

Enfin, la méthode « classique » de construction de systèmes experts nous encourage à dégager les règles empiriques mais ne nous offre aucun moyen pour construire des descriptions structurées du domaine que nous étudions.

Les critiques de Davis ont amené une large réflexion qui a conduit à reconsidérer les techniques traditionnelles de développement de SE. Davis lui-même a décrit une technique bien adaptée aux environnements fortement causaux et bien formalisés, les SE à base de modèles.

16.6 Les systèmes experts par modèles

Nous ne dirons qu'un mot des systèmes experts à base de modèles. Ils sont surtout utilisés pour réaliser du diagnostic de panne. L'idée principale est que « Pour savoir pour-

17 Cette remarque semble enfermer les SE dans un cercle vicieux redoutable : ou l'on utilise une représentation uniforme, et le SE est inefficace, ou l'on utilise des représentations spécialisées et l'on ne peut plus maintenir le programme.

quoi quelque chose est tombé en panne, il est utile de savoir comment il était sensé fonctionner ». La base de notre système expert est donc la connaissance de la structure et du comportement normal du domaine dans lequel nous opérons. En résumé, les systèmes experts par modèle s'appuient sur :

- des langages qui font une distinction nette entre la structure du système à étudier et son comportement, et qui fournissent plusieurs descriptions de la structure, à la fois fonctionnelle et physique ;
- la notion de *chemins causaux d'interaction* qui exprime la relation causale entre les éléments du système.

Davis développe ses idées dans une série d'articles auxquels le lecteur intéressé peut se reporter (Davis 1984; 1987b).

16.7 Mythes et légendes

Dans un récent article (Fox 1990), Mark Fox s'efforce de faire un bilan de ce que les gens croient à propos de l'intelligence artificielle, et en particulier des systèmes experts, en dégageant trois catégories : les mythes, les légendes et les faits. Nous allons reprendre ses conclusions concernant les mythes et les légendes, car, si nous ne les partageons pas complètement, elles représentent ce que les anglais appellent « The Accepted Wisdom[18] » dans le domaine de l'IA (et plus particulièrement de l'ingénierie cognitive).

16.7.1 Les mythes

- **Tous les « systèmes experts » sont des systèmes experts :** comme nous le signalions précédemment, les véritables systèmes experts sont rares. La plupart du temps, la part d'expertise dans le système est faible ou nulle ; on baptise du nom de système expert un peu n'importe quel programme pour des raisons purement commerciales.
- **Les systèmes experts ne font pas d'erreur :** si un système expert émulait véritablement le comportement d'un expert humain, il ferait exactement les mêmes erreurs[19].
- **Les systèmes experts remplacent les approches conventionnelles :** les systèmes experts sont utiles seulement lorsqu'il n'existe pas de techniques algorithmiques adaptées au problème[20]. *Il vaut mieux lorsque cela est possible utiliser des techniques classiques de programmation.* Ainsi, les algorithmes de programmation linéaire sont bien plus efficaces pour résoudre nombre de problèmes d'optimisation que toutes les méthodes expertes.
- **Il suffit d'avoir un expert pour faire un système expert :** c'est une des plus graves erreurs faites en IA, comme le dit très justement l'auteur. Pour écrire un système

---

18 Il est bien difficile de traduire cette expression en français. Il s'agit de l'ensemble des faits, des opinions, reconnus à peu près par tous.

19 En plus des possibles erreurs de codage, oublis, défauts de représentation...

20 Rappelons que pour M. Fox, les techniques algorithmiques de recherche du type programmation linéaire ne sont pas de l'IA. Au demeurant, l'article souffre de l'absence d'une définition précise de l'IA, un sujet, nous l'avons vu dans l'introduction, bien difficile.

expert, il faut que le problème soit clairement identifié et spécifiable. *D'autre part, il faut être conscient que le système expert ne fera que reprendre l'expertise existante (avec toutes les imperfections que suppose cette étape), et que si l'on désire avoir une approche innovatrice, il faudra trouver une autre voie.*

- **Il est facile de passer du prototype au système opérationnel :** le problème est de savoir si la méthode utilisée par le prototype pourra être étendue au problème dans son intégralité. Ainsi, par exemple, nombre de maquettes ou de prototypes de système expert utilisent des techniques dites *de classification*. On sait aujourd'hui que lorsque les connaissances sont fortement interdépendantes, ce type d'approche ne permet pas de passer simplement du prototype à la résolution du cas général. D'autre part, dans bien des cas, *la validité du prototype ne garantit même pas qu'il soit possible de résoudre le problème dans le cas général.*
- **Pour faire de l'IA, il suffit d'apprendre à se servir d'un outil d'IA :** beaucoup de gens pensent qu'en apprenant à se servir d'un outil développé par des spécialistes d'IA, ils seraient à même de mettre en œuvre des techniques d'IA. En fait, apprendre à utiliser un outil d'IA n'apprend rien sur les théories de l'IA, et n'apprend rien non plus sur la méthodologie à appliquer pour écrire un système expert correct.

16.7.2 Les légendes

- **Les systèmes experts sont faciles à écrire :** tout dépend de la forme de la connaissance à coder. S'il existe des techniques d'IA et des outils d'IA déjà sur le marché permettant de résoudre le problème, cela est effectivement facile. En revanche, si nous sommes face à un problème dépendant largement de paramètres locaux, alors le travail est extrêmement complexe.
- **Les systèmes experts sont faciles à valider :** la vérification et la validation de systèmes experts sont probablement bien plus complexes que celles de systèmes traditionnels. En effet, la vérification (qui consiste à garantir que le système final implante bien les spécifications) est difficile à faire car les spécifications des problèmes d'IA sont bien souvent floues. Quant à la validation, elle doit aussi valider le savoir de l'expert, ce qui est bien difficile.
- **Les systèmes experts sont faciles à maintenir :** un des contre-exemples les plus célèbres est R1/XCON de DEC qui, avec plusieurs milliers de règles, dépasse presque les limites raisonnables de la maintenabilité. En fait, la notion de génie logiciel est bien floue en IA, et les systèmes experts sont souvent bien plus difficiles à maintenir que les programmes traditionnels.

16.8 Conclusion

Comme nous venons de le voir, la réalité des systèmes experts est bien différente de l'image que l'on peut s'en faire. Suivant la phrase de Wade Schuette :

> « Best use of "expert systems" is possibly to replace the BOTTOM third of the bell-curve of performance. Forget even "expert". »[21]

En effet, il est probable que la meilleure utilisation des systèmes experts est de remplacer ou d'assister l'homme pour effectuer des tâches peu complexes et répétitives. On peut considérer l'exemple des systèmes experts qui effectuent le réglage de l'exposition sur certains appareils photographiques. Ils leur manquent inévitablement certaines données :

« **syntaxique** » : l'appareil ne connaît que la lumière réfléchie et pas la lumière incidente, par exemple ;

« **sémantique** » : l'appareil ne peut savoir quel est l'objet de la photographie, et donc quelle exposition adopter en cas de contre-jour par exemple.

On rencontre le même type de problèmes dans tous les autres domaines auxquels peuvent s'appliquer les systèmes experts. Il faut donc se garder de prendre le mot *expert* dans son sens littéral.

---

21 « Le meilleur usage des « systèmes experts » est peut-être de remplacer le tiers INFÉRIEUR de la courbe en cloche des performances. Oubliez simplement le mot « expert ». »

Partie IV

Les langages de l'IA

> It gets very frustrating to sell electricity to your customers when they keep asking you for cleaner burning wicks for their oil lamps. The argument shouldn't be what the customer wants but rather if they can do their job enough better with the new paradigm to justify their cost in switching.
>
> — *Martin Fouts*

CHAPITRE 17

Programmation fonctionnelle : ML

Pascal Brisset, Thomas Schiex

17.1 Introduction

Nous limiterons notre introduction à la programmation fonctionnelle au langage Objective Caml (« OCaml » dans la suite), le dialecte français du langage ML. Ce choix a été fait pour de nombreuses raisons :

- parce qu'il fallait inclure – dans un livre dédié (en large partie) à l'intelligence artificielle – une présentation d'un langage fonctionnel, dont le premier représentant de la famille, LISP, a été l'unique langage de l'IA pendant près de vingt ans ;
- parce que l'utilisation d'Objective Caml à travers le monde est en pleine explosion et qu'il est devenu un/le langage de référence pour l'initiation à la programmation en France, aussi bien au sein des classes préparatoires que dans les universités et les grandes écoles ;
- parce que la simplicité et l'élégance de ML et en particulier d'OCaml, permettront un exposé succinct et précis.

Les lecteurs intéressés par les langages fonctionnels typés pourront se reporter avec bonheur à (Paulson 1991; Cousineau and Huet 1989; Leroy and Weis 1993; Weis and Leroy 1993a; Field and Harisson 1988).

La suite de ce chapitre sera formée d'une présentation succincte de OCaml, avec quelques exemples d'utilisation dans le style fonctionnel. Nous donnons ensuite quelques aperçus sur les facilités du langage concernant la modularité, le style impératif et l'extension objet.

17.2 Le langage OCaml

Tout comme la logique du premier ordre constitue la base de la programmation logique, le λ-calcul constitue la source d'inspiration de la programmation fonctionnelle. On peut tracer le rapide historique suivant : dans les années 30, CHURCH invente le λ-calcul, un modèle de calcul qu'il montrera équivalent aux autres modèles informatiques (machine de Turing) et mathématiques (fonctions récursives). En 1958, MAC CARTHY propose de langage LISP (Carthy 1960) dont la sémantique est décrite directement avec le λ-calcul. Dans la même lignée, BACKUS propose en 1978 un langage sans variable

basé sur les combinateur (Backus 1978). La même année Milner (Gordon *et al.* 1978) propose le langage ML comme langage de pilotage d'un démonstrateur automatique. ML est reconsidéré plus tard comme langage de programmation généraliste et décliné en plusieurs dialectes dont les principaux sont :

SML où le « S » signifie Standard (Milner *et al.* 1990) ;

Miranda (Turner 1986a) une version à évaluation paresseuse ;

Caml une implémentation Inria-ENS utilisant la CAM (*Categorical Abstract Machine* (Cousineau *et al.* 1985)). *Caml Light*, une implémentation portable et destinée à des micro-ordinateurs, apparaît en 1990 (et Pierre Weis 1993; Weis and Leroy 1993b). C'est l'ancêtre d'OCaml, la version actuelle distribuée par l'Inria (`caml.inria.fr`)

17.2.1 Un premier survol

Énumérons rapidement les caractéristiques essentielles de ML :

- *portée lexicale* : Tout comme ADA, PASCAL, ALGOL et le λ-calcul, ML est un langage à portée lexicale. Plus simplement, toute occurrence d'une variable est associée à une liaison textuellement apparente de cette variable. En λ-calcul (§ 12.2.1), la portée de l'occurrence liée d'une variable x est l'abstraction de paramètre x dont l'application à M entraînera la substitution de M à x, ou encore, plus informellement, la *liaison* de x à M. Cette caractéristique essentielle n'existe pas dans tous les langages fonctionnels ; de nombreuses implémentation de LISP sont à « liaison dynamique »[1].
- *fonctions : citoyens ordinaires.* La majorité des langages couramment utilisés en informatique distinguent de façon nette les données, passives, des procédures ou programmes, actifs. ML, comme tout langage fonctionnel, permet de considérer les fonctions comme des objets ordinaires du langage. Il est ainsi possible de les manipuler comme des structures de données, de les prendre en paramètre ou de les retourner en valeur[2]. Cette caractéristique se retrouve dans les langages LISP, MIRANDA, HOPE..., à un degré moindre dans COMMON LISP et naturellement dans le λ-calcul (où tout est fonction).
- *fortement typé :* à chaque valeur est associé un type. Le polymorphisme paramétrique est possible mais pas la surcharge (par exemple il existe une fonction + pour les nombres entiers et une fonction +. pour les nombres flottants). Les types ne sont pas déclarés par l'utilisateur mais *inférés* par le compilateur. Le typage fort ne doit pas être considéré comme une contrainte mais comme une aide à la programmation (un chat est un chat).
- *durée de vie correcte* pour tous les objets créés ou alloués durant une exécution. C'est du moins le postulat sur lequel peut s'appuyer avec bonheur le programmeur

---

1 L'occurrence d'une variable n'est plus associée à une liaison textuellement apparente, mais à une liaison qui est définie à l'exécution. Son utilisation, outre qu'elle rend la lecture des programmes plus délicate, puisque le texte d'une fonction ne suffit pas pour savoir où chaque occurrence de variable prendra sa valeur (sera liée), entraîne un grand nombre de problèmes à l'utilisation mais facilite grandement l'implémentation.

2 On parle de facilité d'ordre supérieur, puisqu'il devient ainsi possible de définir des fonctions manipulant des fonctions. De telles possibilités sont aussi présente en programmation logique grâce à des prédicats particuliers.

ML qui n'a pas à se préoccuper de la tâche ingrate (et stupide) de gestion mémoire. Cette gestion est réalisée par un système automatique ou *Garbage Collector* de récupération des objets « inutiles »[3].

- *appel par valeur :* les arguments d'une procédure ML sont toujours évalués avant que la procédure ne soit appelée. Ce calcul a lieu, que la procédure ait, ou n'ait pas, réellement besoin de l'argument. L'appel par valeur est utilisé dans la grande majorité des langages de programmation : C, ML, APL, ADA...). Celui-ci s'oppose à l'appel par nécessité (*call by need*) ou à l'appel par nom (*call by name*) qui pouvait être pratiqué dans ALGOL 60. Le rapprochement avec le λ-calcul est immédiat (§ 12.3.6) : l'appel par valeur correspond grossièrement à la stratégie de réduction non-standard dite *leftmost innermost*, les appels par nom et par besoin correspondent à la stratégie de réduction standard. Ce choix fait perdre la bonne propriété de la réduction gauche (une expression qui possède une forme normale est normalisable par réduction gauche). Ceci peut être comparé à la stratégie « de gauche à droite » de Prolog qui fait perdre la complétude de la résolution.
  Notons enfin qu'OCaml fournit une librairie `Lazy` permettant une évaluation paresseuse des objets pour lequel on le désire (typiquement des objets infinis).
- *traitement des récursivités terminales :* la récursivité est le moyen d'expression naturel de contrôle en programmation fonctionnelle. Le traitement des récursivités dites terminales (*TRO* pour *Tail Recursive Optimization*) permet l'exécution d'itérations en espace constant, même si le calcul a été syntaxiquement décrit par une procédure récursive (dont l'exécution nécessite dans le cas général une pile).
  Exemple : la définition suivante est récursive terminale (l'appel récursif est le dernier appel de la fonction), l'exécution s'effectuera sans consommation de pile.

```
let rec naive_plus a b =
  if a = 0 then b else naive_plus (a - 1) (b + 1);;
```

17.2.2 Syntaxe et typage

Dans un premier temps, la syntaxe de ML peut être rapprochée de la syntaxe du λ-calcul. Il s'agit d'un λ-calcul appliqué faisant donc intervenir en sus (des identificateurs de variables des abstractions et des applications) des symboles de constantes et des nouvelles constructions syntaxiques.

En ML, on manipule des *expressions* :

- `1729` : nombre entier
- `3.14` : nombre flottant
- `"mach31"` : chaîne de caractères
- `function x -> x` : fonction ($\lambda x.x$)
- `fun x y -> x` : fonction à plusieurs arguments ($\lambda xy.x$)
- `truncate 3.15` : application d'une fonction à un argument
- `float ((+) (1) 2)` : expression parenthésée
- `1+2` : opérateur infixe
- `(abs "un")` : application syntaxiquement valide !

---

3 Le critère d'utilité d'un objet dépend de la sémantique opérationnelle de chaque langage. Habituellement, un objet est utile s'il existe un chemin de références depuis la pile d'exécution jusqu'à ce dernier ; cependant ceci n'est pas vrai au sein des langages de programmation logique (Bekkers *et al.* 1984).

Chaque expression est qualifiée par un type (§ 12.6.1) qui, en première approximation, peut être considérée comme un domaine. La notation classique, utilisable au sein du langage, est de la forme *expression* : *type*.

- `1729 : int`
- `3.14 : float`
- `"mach31" : string`
- `function x -> x : 'a -> 'a`
- `truncate : float -> int`
- `(+) : int -> int -> int`
- `1+2 : int`
- `(abs "un")` est mal typé!

où `int`, `float` et `string` sont des types de base. Un type α `->` β désigne une valeur fonctionnelle prenant un argument de type α et retournant une valeur de type β. La fonction `truncate` transforme un nombre flottant en entier (par arrondi). La fonction + prend deux entiers en argument et retourne un entier. La fonction identité est polymorphe : elle prend un argument de type α (à cause de la pauvreté lexicale d'un terminal standard, la variable de type α est notée `'a`) et retourne un résultat de type α.

Le type d'une expression est inféré par le compilateur et il n'est jamais nécessaire à l'utilisateur de le préciser (sauf cas exceptionnel pour l'aide à la mise au point et pour des besoins d'entrées-sorties). Le type doit être compris comme une abstraction de la valeur ; le *typage* d'une expression consiste à un calcul sur ces abstractions, ce dernier étant effectué à *compile time*. C'est ce calcul qui permet de détecter à la compilation certaines erreurs de programmation qui, dans un langage faiblement typé, seraient détectées pendant l'exécution. Le système de types de OCaml est tel que le typage est effectué entièrement à la compilation, i.e. il n'y a pas d'erreur de typage pendant l'exécution du programme.

17.2.3 Évaluation

L'évaluation des expressions est semblable à celle des autres langages de programmation ; la différence essentielle provient de la possibilité d'appliquer *partiellement* une fonction.

- Constantes : `1729` s'évalue en `1729`, `3.14` en `3.14`
- Fonctions : `(fun x -> x)` s'évalue en $\lambda x.x$
- β-réduction : `((fun x -> x) 1)` s'évalue en $(\lambda x.x\ 1)$, puis en `1`
- Primitives : + s'évalue en $\lambda xy.(x + y)$
- Application partielle : `((+) 1)` s'évalue en $\lambda x.(x + 1)$
- Exception : `1 / 0` ; syntaxiquement correct, typé, sans valeur

Toutes les fonctions sont curryfiées : la fonction + (de type `int->(int->int)`) doit être vue comme une fonction prenant un premier argument et retournant une fonction (de type `int->int`) attendant un second argument et retournant un entier. Grâce au fait que les fonctions sont des valeurs ordinaires du langage, l'application partielle d'une fonction curryfiée est possible.

L'ordre d'évaluation d'une application est le suivant : la fonction et les arguments sont évalués dans un ordre **non spécifié**[4] puis la β-réduction (substitution des paramètres de la fonctions par les arguments) est effectuée.

17.2.4 Nommage

La construction `let in` permet d'évaluer une première expression et de la nommer afin de l'utiliser (éventuellement plusieurs fois) au sein d'une seconde expression.

```
let x = truncate 3.15 in x*x*x;;
let square = fun x -> x*x in square 2 + square 3;;
let sum_square = fun x -> fun y ->
  let square = fun x ->
    x*x in
  square x + square y in
sum_square 2 3;;
```

où le `;;` est le séparateur d'expressions.

La syntaxe (simplifiée ici) est : `let` *identificateur* = *expression1* `in` *expression2*.

La sémantique, si on ne tient pas compte du typage est : (λ*identificateur. expression2 expression1*). Par exemple, la première expression de l'exemple est sémantiquement équivalente à `(fun x -> x*x*x) (truncate 3.15)`. Cependant la sémantique opérationnelle des deux expressions n'est pas équivalente, l'ordre d'évaluation n'étant pas nécessairement le même (ce qui n'importe pas pour des expressions fonctionnelles pures possédant une forme normale).

En fait, la différence essentielle entre le `let in` et la sémantique évoquée au paragraphe précédant concerne le typage. On peut comparer les deux expressions suivantes :

```
let id = fun x -> x in (id 1, id "un");;
(fun id -> (id 1, id "un")) (fun x -> x);;
```

où la virgule (`,`) est le constructeur de couple de valeurs.

La première expression est correctement typée alors que la seconde ne l'est pas. La construction `let in` permet de *généraliser* une fonction polymorphe en ajoutant un $\forall\alpha$ devant le type inféré α->α de la fonction identité (§ 12.6.2). Cette quantification est en suite instanciée avec `int` pour la première application et avec `string` pour la seconde. Dans la seconde expression, le typage échoue car la variable de type α de la fonction identité ne peut pas être substituée par `int` **et** par `string`.

Un programme étant constitué par une séquence d'expressions, la construction `let` sans `in` permet de nommer une valeur pour la suite du programme (après le `;;` qui peut donc être vu comme un `in`).

---

4 L'implémentation actuelle évalue les arguments de droite à gauche.

17.2.5 Ordre supérieur

L'exemple de code précédent utilise une facilité originale du langage : une fonction, nommée ou non, peut être argument d'une autre fonction. Cette fonctionnalité permet de définir par exemple la composition de fonction :

```
let compose = fun f -> fun g -> fun x -> f (g x)
```

Cette caractéristique du langage est largement utilisée par les *itérateurs* pour les structures de données : pour un tableau, l'itérateur applique une fonction uniformément à tous les éléments. La définition et l'utilisation d'itérateurs est naturelle en programmation fonctionnelle.

17.2.6 Types de base et conditionnelle

La librairie standard d'OCaml fournit un ensemble de fonctions sur les types de bases : arithmétique sur les entiers et les flottants, opérations sur les booléens (`true` et `false` de type `bool`), manipulation de chaînes de caractères (pour lesquelles l'allocation explicite de mémoire n'est jamais utile). La structure de donnée la plus simple et non atomique est le *couple* dont les deux éléments sont séparés par une virgule (`,`). Un couple d'éléments de type `'a` et `'b` est de type `'a*'b`, le produit cartésien des deux.

Le contrôle de base d'un langage fonctionnel, à la différence des langages impératif, n'est pas la séquence mais la composition fonctionnelle : une itération est naturellement réalisée par une fonction récursive. La conditionnelle est tout de même nécessaire[5] : `if` *si* `then` *alors* `else` *sinon*.

```
let nombre_pair_superieur = fun x ->
  if x mod 2 = 0 then x else x + 1
```

17.2.7 Pattern-matching

Une construction puissante et essentielle du langage ML est le *pattern-matching*. Il est semblable au traitement par cas (`switch` en C) des langages impératifs mais est étendu à la manipulation de données structurées. On peut l'utiliser simplement comme suit :

```
let reponse_oui = fun s ->
  match s with
    "oui" -> true
  | "yes" -> true
  |     _ -> false;;
```

où la valeur de `s`, l'argument de la fonction, est comparée successivement aux *patterns* apparaissant à gauche des flèches (`->`). En cas d'égalité, l'expression correspondante à droite de la flèche est évaluée. Le *pattern* `_` réussit pour n'importe quelle valeur. Un pattern peut également être une variable, qui réussira avec n'importe quelle valeur tentée et qui sera liée à cette valeur :

---

5 L'appel par valeur ne permet pas de définir la conditionnelle : l'appel d'une éventuelle fonction `if` à 3 arguments, (la condition, le alors et le sinon) provoquerait l'évaluation de tous les arguments, indépendamment de la valeur de la condition.

```
let divise1 = fun n -> fun m ->
  match m-1 with
    0 -> failwith "Division par zero"
  | m1 -> n / m1
```

où `failwith` est une fonction de traitement des exceptions que nous ne détaillerons pas ici. Si la valeur v de `m-1` est différente de 0, le pattern `m1` est utilisé et cette variable est liée à v pour la suite du calcul.

La force de ce mécanisme est de faciliter le traitement des données structurées. Voici la fonction qui retourne le premier élément d'un couple :

```
let gauche = fun couple ->
  match couple with
    (a, b) -> a
```

Ici, le pattern réussit si la valeur testée est un couple dont le premier élément peut se comparer avec la variable `a` et le second élément avec `b`, ce qui est toujours le cas puisque `a` et `b` sont des variables (le typage fort assure que la fonction `gauche` sera appliquée uniquement à des couples).

17.2.8 Fonctions récursives

La syntaxe donnée plus haut suffit pour définir des fonctions récursives modulo les deux remarques suivantes : une fonction récursive doit être nommée et le nommage doit être fait avec la construction `let rec`. La définition de la fonction factorielle s'écrit donc :

```
let rec fact = fun n ->
  if n = 0
  then 1
  else n * fact (n - 1)
```

Le déroulement de l'évaluation de l'expression `(fact 2)` se détaille comme suit :

```
 1  fact 2
 2  (fun n->if n=0 then 1 else n*fact (n-1)) 2
 3  if 2 = 0 then 1 else 2 * fact (2 - 1)
 4  2 * fact (2 - 1)
 5  2 * (fun n->if n=0 then 1 else n*fact (n-1)) 1
 6  2 * if 1 = 0 then 1 else 1 * fact (1 - 1)
 7  2 * 1 * fact (1 - 1)
 8  2 * 1 * (fun n->if n=0 then 1 else n*fact (n-1)) 0
 9  2 * 1 * if n = 0 then 1 else 0 * fact (0 - 1)
10  2 * 1 * 1
11  2
```

Tout d'abord, les deux membres de l'application sont évalués (ligne 2) puis l'application est réalisée par substitution de `n` par 2 (ligne 3). Le test de la conditionnelle échoue et le *sinon* est évalué (ligne 4). Le processus se répète jusqu'à terminaison de la récursion (ligne 10). La multiplication peut alors être effectuée (ligne 11).

La récursivité est le moyen privilégié en programmation fonctionnelle pour réaliser des *boucles*.

17.2.9 Listes

La structure de donnée *liste* est à la programmation fonctionnelle ce que le tableau est à la programmation impérative. C'est une structure de donnée récursive dont les traitements sont récursifs.

Une liste est soit vide, soit constituée d'une paire (un *cons*) d'un élément et d'une autre liste. La liste vide est notée `[]` et le constructeur de paire est infixe et noté `::`. La nature récursive d'une liste impose qu'une liste est finie si et seulement si elle se *termine* par la liste vide. La liste des 3 premiers entiers s'écrit `1::(2::(3::[]))` ou `1::2::3::[]`. Enfin, les listes de ML sont homogènes, c'est-à-dire ne contiennent que des éléments de même type.

Construisons la liste des n premiers entiers non nuls dans l'ordre décroissant :

```
let rec premiers_entiers n =
  if n = 0
  then []
  else n :: premiers_entiers (n-1)
```

Le parcours d'une liste se réalise naturellement grâce à une fonction récursive et au pattern-matching :

```
let rec longueur = fun l ->
  match l with
    []    -> 0
  | x::xs -> 1 + longueur xs
```

Le pattern-matching permet ici, d'une part de tester si la liste est vide et d'autre part dans le cas contraire d'*extraire* les deux constituants de la paire, `x` le premier élément de la liste et `xs` le reste de la liste.

À la différence d'un tableau, une liste n'a pas une longueur fixée et il est toujours possible de rajouter un élément en tête. De même, le contenu d'une liste n'est pas modifiable et il n'est pas possible d'accéder directement à n'importe quel élément (pas d'« accès aléatoire »).

17.2.10 Itérateurs

Le traitement des listes étant toujours similaire, il est utile d'abstraire l'opération de parcours en une fonction d'ordre supérieur appelée *itérateur*. Le rôle d'un itérateur est d'appliquer uniformément une fonction aux éléments d'une structure de donnée (ici une liste) et de composer les résultats.

L'itérateur le plus simple pour les listes est sûrement le `map` défini par

```
let rec map = fun f -> fun l ->
  match l with
    []    -> []
  | x::xs -> f x :: map f xs
```

dont le premier argument est une fonction qui est appliquée à chacun des éléments de la liste. Le résultat est la liste des valeurs retournées. Pour ajouter 1 à tous les éléments d'une liste `l` d'entiers on écrira `(map (fun x -> x + 1) l)`.

L'itérateur `fold_right` est l'itérateur canonique associé aux listes. Il permet de *remplacer* les constructeurs de la structure de données (`[]` et `::`) par des fonctions quelconques (`c` et `n`). Pour la liste `1::(2::(3::[]))` (qui s'écrirait `(:: 1 (:: 2 (:: 3 [])))` si le constructeur `::` n'était pas infixe), `fold_right` calcule `(c 1 (c 2 (c 3 n)))`. Il est défini simplement par

```
let rec fold_right = fun c -> fun n -> fun l ->
  match l with
    []    -> n
  | x::xs -> c x (fold_right c n xs)
```

Pour ajouter tous les éléments d'une liste d'entiers `l`, on écrira alors

```
let somme = fun l -> fold_right (+) l 0
```

De même, la fonction `map` précédente peut se définir par

```
let map = fun f -> fun l -> fold_right (fun x r -> f x :: r) [] l
```

ou encore en effectuant une η-réduction

```
let map = fun f -> fold_right (fun x r -> f x :: r) []
```

17.2.11 Définition de types

La richesse du système de type d'OCaml est ouverte à l'utilisateur qui peut (et qui doit) définir de nouveaux types à sa convenance. Ces types utilisateurs se construisent grâce à des ingrédients bien définis et orthogonaux :

- Les types produits : en sus du type produit cartésien (`'a*'b`) prédéfinis, du genre *enregistrement* (`record` en Pascal, `struct` en C).
- Les types sommes : similaires aux unions et énumérations de C.
- Les types récursifs : il n'est pas nécessaire d'utiliser des pointeurs pour exprimer la récursivité.
- Les types polymorphes : les types ont des paramètres comme les fonctions.

Par exemple, le type des arbres binaires dont les feuilles sont de type `'a` se définira par

```
type 'a arbre =
    Feuille of 'a
  | Noeud of 'a noeud
and 'a noeud = { gauche : 'a arbre; droit : 'a arbre }
```

Ce type, récursif et polymorphe (`'a`), est construit avec un type somme (*constructeurs* `Feuille` ou `Noeud`) et un type produit (*étiquettes* `gauche` et `droit`). On peut remarquer qu'une fonction de parcours d'une valeur de ce type a une structure semblable au type : pattern-matching à deux cas et récursivité :

```
let rec nb_feuilles = fun arbre ->
  match arbre with
    Feuille _ -> 1
  | Noeud {gauche = g; droit = d} -> nb_feuilles g + nb_feuilles d
```

L'itérateur canonique pour ce type s'écrit aussi automatiquement en respectant les règles suivantes :

- à chaque constructeur, on associe une fonction, passée en argument ;
- à chaque constructeur est associé un cas dans un pattern-matching ;
- à un type récursif est associé un itérateur récursif ;
- à un type produit est associé une séquence.

```
let rec iter = fun f -> fun n -> fun arbre ->
  match arbre with
    Feuille x -> f x
  | Noeud {gauche = g; droit = d} -> n (iter f n g) (iter f n d)
```

La fonction précédente `nb_feuilles` est donc une instance de cet itérateur :

```
let nb_feuilles = iter (fun _x -> 1) (+)
```

De même le calcul de la liste des feuilles d'un arbre s'écrit simplement avec cet itérateur (`@` est l'opérateur de concaténation de listes) :

```
let feuilles = iter (fun x -> x::[]) (fun g d -> g @ d)
```

Si les feuilles sont des entiers le calcul de leur somme constitue une troisième instance :

```
let somme = iter (fun x -> x) (+)
```

17.2.12 Le compilateur

Le compilateur OCaml peut être utilisé de diverses manières :

- Boucle interactive (`ocaml`) : c'est le mode d'utilisation habituel pour les langages de la famille Lisp et Prolog. Les expressions saisies par l'utilisateur sont analysées, typées, compilées, exécutées et la valeur ainsi que le type du résultat sont imprimés. Ce mode de fonctionnement, bien qu'utile pour l'initiation et l'enseignement, atteint rapidement ses limites pour des programmes importants (plusieurs modules, connexion avec d'autres langages, interfaces graphiques, ...). On l'utilisera pour découvrir le langage mais on devra lui préférer les modes suivants pour une activité de programmation en vraie grandeur.
- Compilation pour une machine abstraite (`ocamlc`) : c'est le mode standard de développement suivant la séquence classique : édition, compilation et production d'un exécutable, exécution. Dans ce mode, le code produit est émulé par une machine abstraite ; l'avantage est qu'un même code produit peut alors être exécuté sur des plateformes différentes. Un outil de mise au point (`ocamldebug`) permet de tracer une exécution et en particulier de *faire marche arrière*.
- Compilation pour un processeur (`ocamlopt`) : c'est un mode analogue au précédent mais le code produit est du code *natif* pour une architecture donnée. L'avantage est l'efficacité accrue du résultat obtenu. Des tests extensifs ont montré que les programmes obtenus ont une efficacité comparable, en temps, aux compilations obtenues avec les compilateurs C (Bagley 2001).

17.3 Exemples pour l'IA

Un langage fonctionnel est un candidat sérieux à la programmation en IA à cause de la richesse des structures de données qu'il offre et la facilité avec laquelle il est possible de les manipuler. Le programmeur étant débarrassé des problèmes d'allocation et gestion de mémoire, il peut se focaliser sur le choix de la structure de donnée adaptée à son problème. Par exemple, dans un langage impératif, l'utilisation d'une liste chaînée est fastidieuse et délicate car il y est impossible de s'abstraire de l'emplacement mémoire occupé par les maillons. Le problème se complique encore quand il s'agit de manipuler des arbres ou des graphes, objets fréquent dans les modèles de l'IA.

Dans cette section, nous illustrons l'utilisation du langage fonctionnel OCaml pour l'implémentation d'un évaluateur de formules logique et d'un A^*.

17.3.1 Évaluation d'une formule propositionnelle

Nous nous proposons ici de représenter les formules de la logique propositionnelle (§ 2) et de les évaluer.

La représentation d'une donnée consiste en la définition d'un type. En suivant les définitions et en choisissant de représenter les variables propositionnelles par des chaînes de caractères, on obtient :

```
type variable = string
type formule =
    Atome of variable
  | Equivalence of formule * formule
  | Implication of formule * formule
  | Disjonction of formule * formule
  | Conjonction of formule * formule
  | Négation of formule
```

Notons que la première déclaration de type n'est pas réellement une définition mais seulement le renommage (alias) d'un type existant.

Avec cette représentation, le prédicat (fonction à résultat booléen) qui teste si une formule est un littéral se définit comme suit :

```
let litteral = fun f ->
  match f with
    Atome _             -> true
  | Négation (Atome _)  -> true
  | _                   -> false
```

On note dans cette exemple l'usage du pattern-matching avec un pattern sophistiqué à deux constructeurs : on teste en même temps que la formule est une négation **et** que c'est la négation d'un atome. Dans un pattern, la variable anonyme, qui « matche » n'importe quoi, est notée avec le souligné (_). Le dernier cas de ce `match` correspond donc au `default` du traitement par cas (`switch`) du langage C.

L'écriture d'un interpréteur de formule demande peu d'efforts : il s'agit de réaliser une correspondance entre notre représentation et les connecteurs logiques fourni par le langage OCaml. La gestion des variables est effectuée grâce à un *environnement* composé de paires (identificateur, valeur). La fonction `List.assoc` de la librairie standard[6] permet d'accéder à cet environnement.

```
let rec inter = fun env -> fun formule ->
  match formule with
    Atome id              -> List.assoc id env
  | Equivalence (f1, f2) -> inter env f1 = inter env f2
  | Implication (f1, f2) -> not (inter env f1) || inter env f2
  | Disjonction (f1, f2) -> inter env f1 || inter env f2
  | Conjonction (f1, f2) -> inter env f1 && inter env f2
  | Négation f           -> not (inter env f)
```

Le profil de cette fonction est donc le suivant :

```
inter : (variable * bool) list -> formule -> bool
```

17.3.2 Codage d'un algorithme A^*

Nous reprenons ici les notations de la description donnée en 13.5.6. Les ensembles G et D sont représentés par des listes (OCaml offre des librairies de structures de données plus efficaces pour représenter des ensembles mais ça n'est pas l'objet de l'exposé). Les éléments de G et D sont des listes de quadruplets contenant l'état (u), la coût depuis la racine ($g(u)$), l'évaluation du nœud ($h(u)$) et le chemin depuis la racine. On suppose que nœud u est solution si et seulement si $h(u) = 0$. Le chemin est la liste des nœuds du nœud courant vers la racine. La liste G est triée par ordre croissant selon f.

La fonction `a_etoile` retourne le chemin vers la solution trouvée et prend quatre arguments : l'état initial (u_0), la fonction heuristique (h), la fonction de coût (k) et une fonction permettant de calculer les fils d'un état (la fonction p_i). On suppose que cette dernière fonction retourne une liste d'états. Le type inféré pour `a_etoile` est donc

```
'a -> ('a ->int) -> ('a -> 'a ->int) -> ('a ->'a list) -> 'a list
```

où `'a` est le type d'un état.

Le programme[7] est donné en figure 17.1. La boucle générale est codée par la fonction (locale) récursive `tant_que` dont les deux arguments sont G et D. La boucle de traitement des fils de l'état courant est réalisée par la fonction `boucle_fils` qui prend en argument une liste d'états à insérer (`vs`) et G. Est retourné le nouveau G. Deux fonctions auxiliaires sont nécessaires : `inserse_selon_f` qui insère l'état v à sa place dans la liste G et `deja_vu_et_moins_bon` qui teste la condition d'insertion du fils v. La définition locale des fonctions auxiliaires évite des passages de paramètres fastidieux : par exemple les valeurs de `u`, `gu`, `cu`, ... ne sont pas passées à la fonction `boucle_fils` car celle-ci est définie dans un contexte ou ces valeurs sont connues[8].

---

6 La convention de nommage en OCaml est de préfixer le nom d'une valeur par le nom du module où elle est définie. Il s'agit donc ici de la fonction `assoc` définie dans le module `List`.

7 En utilisant des facilités syntaxiques du langage non détaillées ici et des itérateurs à la place des fonctions récursives, il est facile de réduire de 20% la taille du code.

8 Notons que l'usage extensif des fonctions locale est un compromis entre lisibilité et concision.

```
let a_etoile = fun u0 -> fun h -> fun k -> fun fils ->
  let rec tant_que = fun g -> fun d ->
    match g with
      [] -> failwith "no solution"
    | (u,gu,hu,cu)::rest_g ->
        if hu = 0   (* Heuristique à 0 *)
        then cu     (* Un état solution, cu est son chemin *)
        else begin
          let rec boucle_fils = fun fils -> fun g ->
            match fils with
              [] -> g
            | v::vs ->
                let gv = gu + k u v and hv = h v and cv = u::cu in
                let rec insere_selon_f = fun g ->
                  match g with
                    []                -> [(v,gv,hv,cv)]
                  | (w,gw,hw,cw)::gs ->
                      if gw + hw < gv + hv
                      then (w,gw,hw,cw)::insere_selon_f gs
                      else (v,gv,hv,cv)::(w,gw,hw,cw)::gs
                in
                let rec deja_vu_et_moins_bon = fun d ->
                  match d with
                    [] -> false
                  | (w,gw,hw,cw)::ds ->
                      v = w & gw < gv || deja_vu_et_moins_bon ds
                in
                if deja_vu_et_moins_bon d
                then boucle_fils vs g
                else boucle_fils vs (insere_selon_f g)
          in
          tant_que (boucle_fils (fils u) rest_g) ((u,gu,hu,cu)::d)
        end
  in
  tant_que [(u0, 0, h u0, [])] []
```

Figure 17.1 – Une implémentation naïve de l'algorithme A*

17.4 Aller plus loin avec OCaml

Nous avons donné un rapide aperçu du langage OCaml qui n'est bien entendu pas exhaustif. Bien que concis (313 pages), le manuel de référence (Leroy 1998) décrit exhaustivement la syntaxe (riche) et la sémantique du langage ainsi qu'une librairie bien fournie (tableaux, cryptographie, *pretty-printing*, analyse syntaxique, tri, files, piles, ensembles, système, ...). Nous donnons ici quelques autres fonctionnalités du langage.

17.4.1 Modularité

OCaml propose un système de modules indépendant du langage. Un module est décrit par une interface (la spécification) et une implémentation. Un fichier source (`.ml`) est un module associé à une interface (`.mli`), ceci permet une compilation séparée des différents modules. Un module contient des définitions de type et des valeurs qui sont *privées* au module sauf si elles sont déclarées dans l'interface.

Le système de modules est lui aussi fonctionnel : la définition de *foncteurs* permet de faire dépendre un module d'autres modules. La réutilisabilité du code en est accrue.

Ces mécanismes sont en fait indépendants du système de fichiers et une programmation modulaire est possible au sein d'un même fichier.

17.4.2 Style impératif

La programmation impérative (séquentielle) est indispensable dans n'importe quel langage de programmation pour plusieurs raisons :

- les fonctions d'entrées-sorties ou plus généralement à effet de bord doivent être exécutées dans un ordre fixé ;
- certains algorithmes nécessitent des structures de données modifiables (par exemple des tableaux) pour pouvoir être implémentés correctement.

OCaml possède des structures de données et de contrôle analogues à celles des langages impératifs :

- la séquence : une fonction sans valeur significative (de type `unit`, ce type ne contenant que la valeur `()`) peut être exécutée en séquence avec une autre (`;`) ;
- la boucle `while` : permet d'exécuter répétitivement en séquence ;
- la référence : est une valeur modifiable ; elle est analogue à une variable dans un langage impératif (l'analogie est cependant dangereuse car il n'y a pas un emplacement mémoire associé à une référence) ;
- le tableau : structure de données à n éléments (de même type), à accès aléatoire, et modifiable en place ;
- la boucle `for` : structure de contrôle adaptée au traitement des tableaux.

Par exemple, la fonction qui affiche la factorielle d'un nombre pourra s'écrire en style impératif :

```
let fact = fun n ->
  let p = ref 1 in
  for i = 1 to n do
    p := !p * i
  done;
  print_int !p; print_newline ();;
```

où `ref` permet de construire une référence et `!` d'accéder à sa valeur.

17.4.3 Programmation orientée objet

La contribution la plus récente d'OCaml est la conciliation de la programmation fonctionnelle avec la programmation orientée objet. Cette extension a été faite en conservant les fondements du langage, typage fort inféré et absence de surcharge[9].

OCaml fournit les fonctionnalités habituelles des langages orientés objet : définition de classe, instance (objet), variable d'instance, méthode, héritage, classes paramétriques (*template*), héritage multiple, méthodes et classes virtuelles. Du point de vue typage, l'extension est simple pour l'utilisateur : le type d'une classe est exactement la collection des types de ses méthodes.

9 On parle de surcharge dans un système de type quand qu'une même fonction possède plusieurs implémentations sélectionnées en fonction du type des arguments. A l'heure actuelle, la surcharge n'est pas compatible avec l'inférence automatique dans le cas général (Dubois *et al.* 1995).

CHAPITRE 18

Programmation logique : PROLOG

Jean-Marc Alliot — Thomas Schiex

18.1 Présentation

PROLOG signifie **PRO**grammation en **LOG**ique. Le nom du langage est significatif. Programmer en PROLOG, c'est exprimer sous forme de relations logiques les différentes propriétés d'un système. Une fois le problème formalisé, le langage sera, le plus souvent, capable de fournir les réponses aux questions posées par l'utilisateur en utilisant une méthode de résolution automatique.

La programmation en PROLOG est assez différente de la programmation dans un langage classique. PROLOG est un langage dit *déclaratif*, en ce sens qu'on ne décrit pas comme dans les langages traditionnels (dits *procéduraux*) une méthode (ou algorithme) de résolution, mais que l'on déclare les propriétés du problème, le système se chargeant lui-même d'appliquer une méthode de résolution.

Les formules logiques que PROLOG peut utiliser sont les clauses de Horn de la logique des prédicats telles que nous les avons décrites au chapitre 6. PROLOG met en application les techniques de résolution et d'unification que nous avions présentées formellement dans ce même chapitre. Nous en donnons ici une présentation moins formelle.

PROLOG a vu le jour en 1971 sur une idée d'Alain Colmerauer qui travaillait alors aux problèmes de traduction automatique. Le premier langage PROLOG a été créé par Alain Colmerauer avec l'aide de Philippe Roussel dans le cadre d'un contrat de recherche sur la communication homme-machine. La syntaxe de ce premier PROLOG est assez différente de ce qu'elle sera par la suite ; la notion de *cut* n'existait pas encore ; en revanche, le problème du test d'occurrence était traité, et il était possible de pratiquer l'évaluation différée. Il était écrit en Algol-W sur IBM 360-37. Pour la petite histoire, le nom PROLOG fut inventé par la femme de Philippe Roussel.

PROLOG I fut développé à partir du PROLOG original dès 1972. Philippe Roussel introduisit, sur une idée de Boyer et Moore, la technique de partage de structures, la syntaxe fut modifiée et la forme des clauses se rapprocha de celle que nous connaissons aujourd'hui. PROLOG I fut implanté pour la première fois en FORTRAN, en 1973. La restriction aux clauses de Horn ainsi que la sémantique de PROLOG furent établies par R. Kowalski et M. van Emden en 1974 (Kowalsi and van Emden 1974). Le manuel de PROLOG I dans sa version « définitive » fut terminé en 1975.

C'est en 1974 que David Warren s'intéresse pour la première fois à PROLOG pour la programmation de son système Warplan. En 1977, il développe le premier « compilateur » PROLOG (Warren 1977) avec Fernando Pereira et Luis Moniz Pereira (Pereira *et al.* 1979). Il introduira la syntaxe dite « syntaxe Edimburgh », reprise par la plupart des PROLOG aujourd'hui.

À partir de 1978 les implantations de PROLOG vont se multiplier, en Angleterre à l'université d'Edimburgh et à l'Imperial College de Londres, en Belgique avec les travaux de G. Roberts et M. Bruynooghe, en Hongrie avec Peter Szeredi.

C'est en 1983 que David Warren définira la machine abstraite qui porte son nom (Warren 1983). Il s'agit d'une amélioration de son travail sur le compilateur fait en 1977. Cette machine reste aujourd'hui la référence en matière de compilation PROLOG. Elle a donné naissance à QUINTUS-PROLOG, SB-PROLOG, SWI-PROLOG et bien d'autres implantations[1].

En France, PROLOG II a été développé à la suite de PROLOG I par l'équipe d'Alain Colmerauer, et plus particulièrement implanté par Michel van Caneghem dans les années 1980-1983 (van Caneghem 1986). Il s'agit de la première version véritablement commerciale des PROLOG marseillais et elle connaîtra une grande diffusion en France dans les universités et les centres de recherche.

Depuis ces temps héroïques, les implantations de PROLOG se sont multipliées et la recherche s'est divisée en plusieurs sous-domaines : compilation, implantations parallèles, algorithmes d'unification par contraintes, etc. Les décrire toutes demanderait probablement une encyclopédie. On peut se reporter pour plus de précisions à (Boizumault 1988).

Nous nous basons pour présenter PROLOG sur le système PROLOG II de l'équipe d'Alain Colmerauer, équipe qui fut à l'origine du développement de PROLOG au début des années 70. Il existe de nombreuses autres variantes de PROLOG dont PROLOG-EDIMBURGH et surtout, pour le développement de gros systèmes experts, QUINTUS-PROLOG (dont la syntaxe est celle de PROLOG-EDIMBURGH). Nous ne ferons pas une présentation détaillée de PROLOG ; le lecteur désirant utiliser le langage peut se rapporter à la nombreuse littérature existante dont, par exemple, (Giannesini *et al.* 1985) pour PROLOG II ou (Bratko 1988; Sterling and Shapiro 1986) pour C-PROLOG ou QUINTUS-PROLOG.

18.2 Terminologie PROLOG

Nous décrivons rapidement l'emploi de certains termes dans le contexte PROLOG :

base de faits : on appelle *base de faits* l'ensemble des faits relatifs au problème que l'utilisateur formalise en PROLOG. « Médor est un chien » peut se représenter par un fait. Formellement, les faits sont des clauses de Horn positives ;

base de règles : on appelle *base de règles* l'ensemble des règles relatives au problème que l'utilisateur formalise en PROLOG. « Un chien est un mammifère » peut se représenter par une règle. Les règles sont des clauses de Horn strictes ;

---

1 Pour bien comprendre la machine de Warren, on peut se reporter à (Ait-Kaci 1991) ou (Gabriel *et al.* 1985) qui sont d'excellentes introductions.

question : la question est une clause de Horn négative. C'est ce que l'utilisateur demande au système ;

moteur d'inférence de PROLOG **:** le moteur d'inférence de PROLOG est la partie du système qui réalise les inférences logiques sur les faits et la question posée par l'utilisateur, de façon à donner une (ou plusieurs) réponses. Il s'agit d'un moteur non-monotone à chaînage arrière et à régime par tentatives, comme nous allons le voir. Il opère sur un ensemble de faits et de règles, donc un ensemble de clauses de Horn négatives ou strictes. Sur un plan théorique, le moteur PROLOG fait une preuve d'inconsistance sur la base de clauses de Horn, comme le permettent différentes méthodes vues aux chapitres 6 et 7.

18.3 La syntaxe

Nous allons présenter la syntaxe PROLOG II très informellement. Cet ouvrage n'étant pas un cours de PROLOG II, notre seul but est de donner au lecteur les bases suffisantes pour comprendre la suite de ce chapitre.

Les variables : une variable est représentée par une seule lettre suivie d'un ou plusieurs chiffres.

```
x
x1
a3
```

sont des symboles de variables autorisés en PROLOG II.

Les symboles de prédicats : un symbole de prédicat est un mot d'au moins deux lettres successives (*Attention : il s'agit de deux lettres, pas de chiffres).* Il peut avoir des arguments, auquel cas il est suivi d'une parenthèse ouvrante, de la liste des arguments séparés par des virgules et d'une parenthèse fermante. Les arguments peuvent être des variables comme `x`, des symboles de fonctions avec leur argument comme `fonc(x)` ou des symboles de constantes comme `toto` par exemple. Les symboles de fonctions et de constantes ont la même structure syntaxique que les prédicats, ce qui renforce l'ambiguïté du langage et encourage les confusions.

```
chien(x)
toto(a,b,c)
truc(fop(x),y).
```

sont des utilisations de symboles de prédicats correctes.

Les listes : une liste est une suite d'objets séparés par des `.` et terminée par `nil`. Les parenthèses sont utilisées pour délimiter les sous-listes :

```
a.b.toto.2.nil
a.(b.c).d.nil
```

sont des listes syntaxiquement correctes en PROLOG II.

Attention, les listes :

```
a.(b.c).d.nil
```

et

```
a.b.c.d.nil
```

ne représentent pas les mêmes listes[2].

Les faits : un fait s'écrit :

```
predicat_tete ->;
```

Un exemple de fait correctement exprimé est :

```
chien(fido)->;
```

Ce qui peut signifier *fido est un chien.*

Les règles : une règle s'écrit :

```
pred_tete -> pred1 pred2 pred3 ... predn;
```

Un exemple de règle correctement exprimée est :

```
mammifere(x) -> chien(x);
```

Ce qui peut signifier *Pour tout x, si x est un chien alors x est un mammifère. Attention : le signe -> n'est pas un signe d'implication.* Il peut s'interpréter comme un signe de réécriture. En logique, nous écririons la dernière clause : chien(x) $\rightarrow$ mammifere(x). Là encore, la syntaxe du langage peut induire en erreur. (Rappelons que `pred_tete` est un littéral positif et que `pred1`, `pred2`,...,`predn` sont des littéraux négatifs).

18.3.1 Les notions de OU et de ET

Les clauses de Horn représentent de façon triviale les notions intuitives de OU et de ET. Un ensemble de clauses $\{C_1, C_2, \ldots, C_n\}$ est équivalent à $C_1 \wedge C_2 \ldots \wedge C_n$. De même toute clause de Horn C est de la forme $(r_1 \wedge r_2 \ldots r_m) \rightarrow h$. Ainsi, soit le programme PROLOG :

```
grandpere(x,y) -> pere(x,z)  pere(z,y);
grandpere(x,y) -> pere(x,z)  mere(z,y);
```

Il représente la phrase : « (x est le grand-père de y si ((x est le père de z ET z est le père de y) OU (x est le père de z ET z est la mère de y)) ».

18.3.2 Premier exemple d'exécution

Voici un premier exemple de programme PROLOG :

```
chien(fido)->;
mammifere(X) -> chien(X);
```

La première ligne est un fait qui déclare que `fido` est un `chien`. La seconde ligne est une règle qui déclare que tout `chien` est un `mammifere`. Une question que l'on pourrait poser serait :

```
? mammifere(fido);
```

---

2 Pour être tout à fait précis, l'opérateur . correspond au `cons` de LISP. Le `nil` placé traditionnellement à la fin des listes correspond à la valeur `nil` terminant une liste LISP. La liste `a.b` (sans le `nil`) en PROLOG II correspond à la valeur de `(cons 'a 'b)` en LISP, alors que la liste `a.b.nil` correspond à la valeur de `(list 'a 'b)`.

et la réponse du programme serait (miracle de l'informatique) :

```
Ok.
```

Si l'on posait la question `? mammifere(x)`, la réponse serait : `x=fido`.

18.4 Moteur d'inférence

Nous allons détailler le fonctionnement du moteur d'inférence PROLOG. Nous allons tout d'abord examiner le mécanisme de résolution en nous réduisant au calcul propositionnel, puis nous discuterons du mécanisme d'unification en calcul des prédicats.

De par son but, PROLOG n'est pas très éloigné de l'algorithme de Davis et Putnam, puisqu'il démontre également l'inconsistance d'une base de clauses de Horn. L'algorithme utilisé s'appuie sur le principe de résolution (cf. §7.6).

18.4.1 Fonctionnement simplifié

Les étapes de fonctionnement du moteur PROLOG sont les suivantes :

1. On prend dans la question courante (qui est une clause de Horn négative, donc ne contenant que des littéraux négatifs) le *premier* littéral négatif.
2. On parcourt la base de clauses (faits et règles) *séquentiellement* depuis la *première* jusqu'à ce que l'on ait trouvé une clause dont le littéral positif, appelé aussi littéral de tête de la clause, est le même que le littéral négatif que nous avons sélectionné. Si l'on ne peut plus en trouver, la résolution est terminée et l'on a échoué.
3. On élimine de l'ensemble des littéraux négatifs qui composaient la question le littéral négatif sélectionné, et l'on rajoute l'ensemble des littéraux négatifs de la clause sélectionnée (ensemble qui peut être vide si cette clause est une clause de Horn positive. Le nouvel ensemble ainsi constitué est la nouvelle question.
4. Si la question est réduite à la clause vide, la résolution est terminée et l'on a réussi. Sinon, on revient à l'étape 1.

Nous allons examiner ce mécanisme sur un exemple ; soit le programme :

```
titi -> toto;
tutu ->;
toto ->;
```

Supposons que la question soit :

```
? titi tutu;
```

Examinons en détail la résolution :

1. Le moteur d'inférence va sélectionner le premier littéral négatif de la question : `titi`.
2. Puis il va chercher dans la base de faits la première clause dont la tête (le littéral positif) est égale à `titi`. Il s'agit de `titi -> toto ;` .

3. Il va éliminer de la question le littéral négatif `titi` et ajouter le littéral négatif (il n'y en a qu'un ici) de la clause sélectionnée : `toto`.
4. La nouvelle question est maintenant : `toto tutu`. Elle n'est pas réduite à la clause vide, donc le système repart à l'étape 1.
5. Il sélectionne dans la question le premier littéral négatif (`toto`).
6. Il sélectionne la première clause dont le littéral positif est égal à `toto`. Il s'agit de `toto -> ;`.
7. Il élimine de la question le littéral négatif `toto`, et ne rajoute rien puisqu'il n'y a pas de littéral négatif dans la clause sélectionnée.
8. La nouvelle question est désormais `tutu`. Le système va répéter les quatre étapes pour arriver cette fois-ci à la clause vide. La résolution est terminée.

18.4.2 Le retour-arrière et la notion de points de choix

La description précédente a été volontairement simplifiée. En effet, considérons le programme suivant :

```
titi -> toto;
titi -> ;
```

Essayons de résoudre la question `titi`. Que se passe-t-il ? Nous sélectionnons le littéral négatif de la question `titi`, puis la première clause utilisable `titi -> toto ;` Maintenant nous éliminons de la question `titi` et nous rajoutons le littéral négatif de la clause, c'est-à-dire `toto`. Et nous ne pouvons plus continuer la résolution. Or il est clair que nous pouvons avec une semblable base de faits déduire `titi` ! Pourquoi échouons-nous ? Visiblement parce que nous avons sélectionné la mauvaise clause `titi -> toto ;` alors qu'il fallait choisir `titi -> ;`. Il faudrait donc pouvoir revenir en arrière jusqu'au choix de cette clause et en choisir une autre. Ce mécanisme s'appelle le *retour-arrière* ou *back-track* en anglais. Les points de la résolution où nous pouvons revenir pour choisir une autre clause s'appellent les *points de choix*.

L'algorithme complet est présenté sur la figure 18.1.

Ainsi, nous devons mémoriser des points de choix et nous devons aussi mémoriser les *contextes* associés à ces points de choix. Un contexte est l'ensemble des informations nécessaires au fonctionnement du moteur d'inférence, à un instant donné. Lorsque nous faisons un retour-arrière, nous devons obligatoirement nous remettre dans l'état exact où nous étions au moment où nous avons choisi une clause. Cela implique donc que nous ayons mémorisé le contexte au moment du choix pour pouvoir le restaurer. Un élément du contexte est par exemple l'ensemble des littéraux négatifs qui constituaient la question au moment où nous avons choisi notre clause. Dans la suite de la résolution, cet ensemble a évolué, et il faut que nous retrouvions l'ensemble des littéraux négatifs tel qu'il était au moment du choix, ce qui implique que nous l'ayons mémorisé.

Nous allons voir ce mécanisme sur un exemple ; reprenons le programme précédent :

```
titi -> toto;
titi -> ;
```

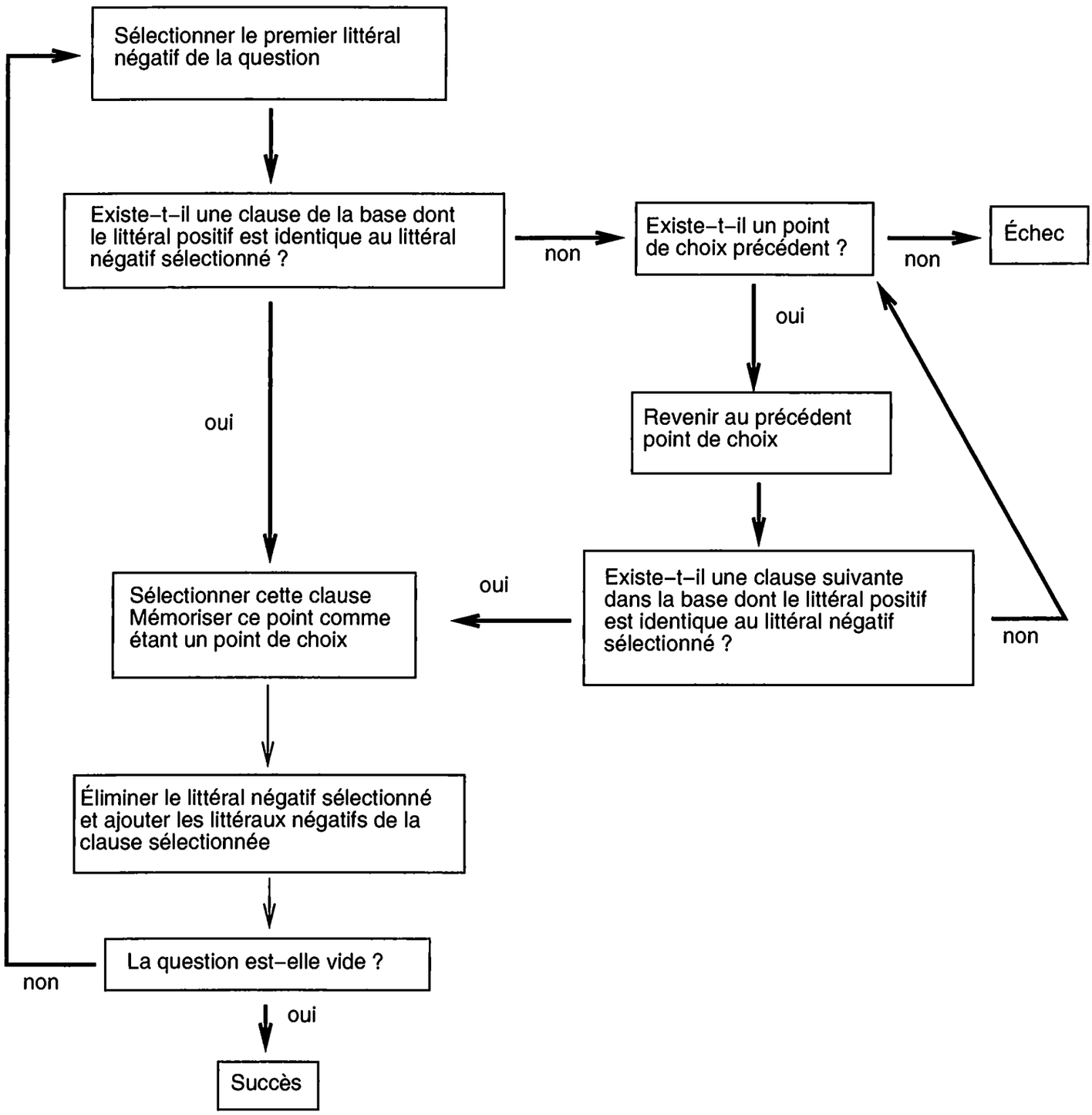

Sélectionner le premier littéral négatif de la question
Existe-t-il une clause de la base dont le littéral positif est identique au littéral négatif sélectionné ?
non
Existe-t-il un point de choix précédent ?
non
Échec
oui
oui
Revenir au précédent point de choix
Existe-t-il une clause suivante dans la base dont le littéral positif est identique au littéral négatif sélectionné ?
oui
non
Sélectionner cette clause
Mémoriser ce point comme étant un point de choix
Éliminer le littéral négatif sélectionné et ajouter les littéraux négatifs de la clause sélectionnée
La question est-elle vide ?
non
oui
Succès

Figure 18.1 – Fonctionnement de l'algorithme PROLOG

Examinons comment le système résout la question `titi`.

1. On sélectionne la première clause dont la tête est égale à `titi`. Il s'agit de `titi -> toto`.
2. On mémorise le point de choix et on sauvegarde le contexte (la valeur de la question courante, en particulier).
3. On construit la nouvelle question qui est `toto`.
4. On ne parvient pas à résoudre la nouvelle question, car on ne trouve pas de clause dont la tête est `toto`. On exécute donc un retour-arrière jusqu'au point de choix précédent, et l'on restaure le contexte (l'ancienne question, `titi`).
5. On sélectionne la clause suivante dont la tête est `titi` : la clause `titi -> ;`.
6. On mémorise ce point de choix.
7. On construit la nouvelle question. Elle est vide, donc la résolution est terminée.

18.4.3 La résolution : un parcours d'arbre ET-OU

Nous avons présenté PROLOG comme un système réalisant des preuves d'inconsistance dans une base de faits. Mais il est également possible de voir PROLOG comme un système faisant du parcours d'arbre ET-OU. Nous allons examiner cela sur un exemple. Soit le programme :

```
titi -> toto tutu;
titi -> tata toto;
tata -> toto;
tata -> tutu;
toto ->;
```

Nous allons voir graphiquement comment le système résout la question `titi`. Nous devons rappeler rapidement le formalisme des graphes ET-OU vus dans la section 13.4.3 : les arcs partant d'un nœud et reliés entre eux sont des arcs ET. Les arcs non reliés sont les arcs OU. Sur la figure 18.2, on voit comment le système, partant de la question, parvient à la solution ; pour résoudre `titi`, le système a le choix entre deux clauses. L'une impose la résolution des sous-buts `toto` et `tutu`, et l'autre la résolution des sous-buts `tata` et `toto`. Le moteur d'inférence réalise un parcours du haut vers le bas et de la gauche vers la droite de l'arbre :

1. Il résout d'abord le sous-but `toto` situé à gauche. Ce sous-but se résout lui-même trivialement.
2. Il reste à résoudre l'autre sous-but de cet arc ET, c'est-à-dire `tutu`. Or, c'est un échec car il n'existe aucune clause permettant de résoudre `tutu`. Le système revient donc au point de choix précédent : le nœud le plus proche contenant un arc OU.
3. Le moteur essaie donc de résoudre les sous-buts `tata` et `toto`.
4. Le sous-but `tata` peut se résoudre de deux façons (arc OU). Soit en résolvant `toto`, soit en résolvant `tutu`. Le sous-but `toto` se résout trivialement. Le sous-but `tata` est donc résolu, et la résolution de `tutu` n'est même pas examinée.
5. Le sous-but `toto` restant est résolu de façon triviale.

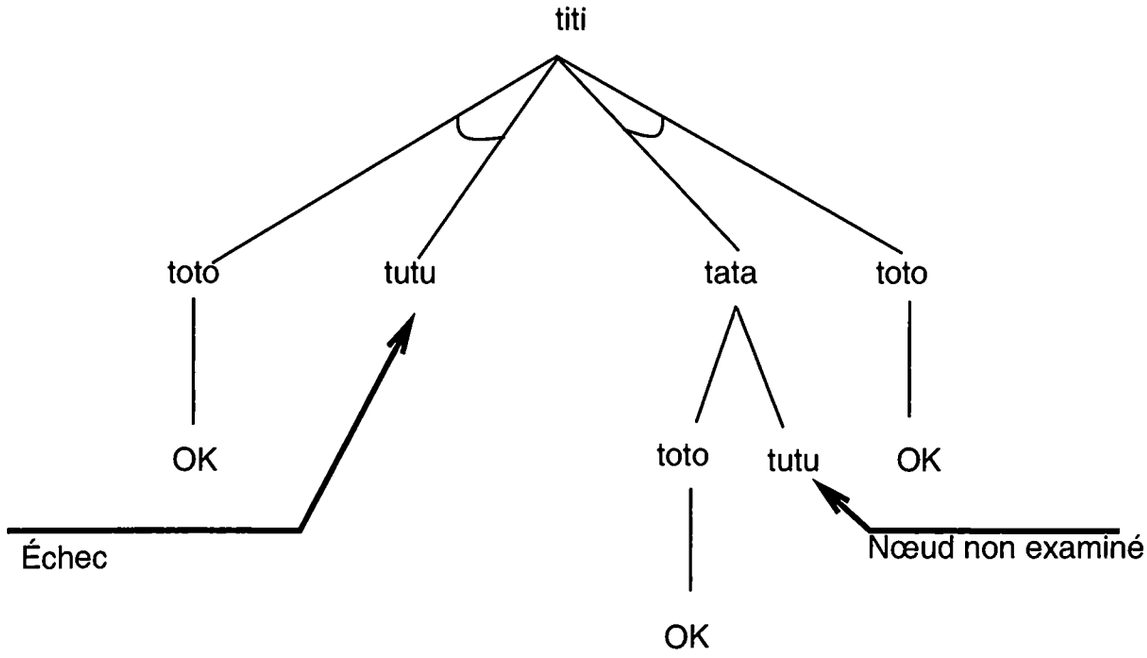


Figure 18.2 – Résolution PROLOG sous forme d'arbre ET-OU

On voit donc que l'exécution d'un programme PROLOG peut être représentée très simplement sous forme d'arbre, et que cet arbre est parcouru de façon systématique : du haut vers le bas et de la gauche vers la droite. C'est cette technique de parcours d'arbre qui est l'*interprétation procédurale* de PROLOG. En fait, bien que PROLOG soit, en principe, un langage déclaratif où le programmeur ne devrait pas avoir à se soucier du *comment* de la résolution, le succès de PROLOG s'explique par le fait que l'on peut facilement interpréter ses mécanismes de contrôle, et modifier ce contrôle.

En fait, si l'on compare à l'algorithme de Davis et Putnam (voir sections 6.3.4, 6.5.2), l'algorithme PROLOG, hormis cet aspect contrôle, est plutôt moins élégant. En effet, l'algorithme de Davis et Putnam garantit la terminaison de la résolution en calcul propositionnel (l'ensemble des littéraux est réduit d'un terme à chaque pas de la résolution), ce qui n'est pas le cas de PROLOG qui peut facilement entrer dans des boucles infinies[3], même dans le cas propositionnel. Nous allons rapidement examiner cela sur un exemple. Soit le programme :

```
titi -> toto;
toto -> titi;
titi ->;
```

Si nous tentons de résoudre la question `titi`, le moteur d'inférence va tenter de résoudre `toto` qui va lui-même tenter de résoudre `titi`... Le moteur rentre dans une boucle infinie dont il ne peut sortir. En revanche, l'algorithme de Davis et Putnam résout aisément ce problème. Partons de l'ensemble de clauses de Horn équivalentes au programme ci-dessus, soit : $\{titi \vee \neg toto, toto \vee \neg titi, titi\}$. On adjoint la question $\{\neg titi\}$, soit : $\{titi \vee \neg toto, toto \vee \neg titi, titi, \neg titi\}$. L'algorithme de Davis et Putnam nous dit de choisir une clause positive (la seule clause positive est $titi$), et une clause de Horn contenant $\neg titi$. Il n'y en a qu'une : $toto \vee \neg titi$. Après réduction, le nouvel ensemble de

3 L'algorithme de Davis et Putnam est d'ailleurs souvent utilisé pour effectuer des preuves de complétude.

clauses est : $\{titi \vee \neg toto, toto, titi, \neg titi\}$. On peut sélectionner cette fois-ci *toto*, et $titi \vee \neg toto$. L'ensemble réduit est alors : $\{titi, toto, \neg titi\}$. Après la sélection de *titi* et $\neg titi$, on est ramené à $\{titi, toto, \varnothing\}$ qui est inconsistant.

Ainsi, l'algorithme de Davis et Putnam permet de donner les meilleurs résultats que le moteur d'inférence PROLOG dans le cas propositionnel, tout en étant aussi efficace (pour la complexité de l'algorithme) et plus général.

Il nous faudra donc examiner ce qui a fait le succès de PROLOG : la possibilité de réaliser des opérations extérieures à la logique pure. Nous le ferons dès que nous aurons présenté le fonctionnement du moteur d'inférence en logique des prédicats.

18.4.4 L'unification

Nous nous sommes jusque-là intéressés uniquement à des prédicats ne contenant pas de variable. Nous allons maintenant nous intéresser à la façon dont PROLOG gère les variables. Considérons le programme suivant :

```
chien(fido) ->;
mammifere(x) -> chien(x);
```

Supposons que la question soit :

```
? mammifere(y);
```

La variable y dans la question initiale est une variable dite *libre*; pour l'instant elle n'a reçu aucune valeur.

Le moteur d'inférence va tout d'abord procéder comme précédemment. Il va choisir la seule clause dont la tête est unifiable (Cf. section 7.5.2) avec la question : `mammifere(x) -> chien(x)`. Il va alors unifier la tête de la clause avec la question, il va donc *lier* les variables y et x. À partir de ce moment-là, y n'est plus une variable libre puisqu'elle prendra toujours la valeur de la variable x. En revanche x est encore une variable libre.

Le moteur d'inférence va alors choisir une clause unifiable avec `chien(x)` qui est la nouvelle question courante. Il n'y a que `chien(fido)` qui soit éligible. Le moteur d'inférence va donc unifier `chien(x)` et `chien(fido)`. La variable x va donc se retrouver liée à la constante `fido`, et, par transitivité, la variable y va elle aussi être liée à `fido`. Le moteur d'inférence affichera alors : `y=fido`

Un problème se pose lors d'un retour-arrière. Ainsi, considérons le programme :

```
grandpere(x,y) -> pere(x,z)  pere(z,y);
pere(jean,marc)->;
pere(eric,jean)->;
```

Et la question `grandpere(u,v)`. Sur la figure 18.3, nous voyons comment le moteur d'inférence résout le problème :

1. le moteur sélectionne la clause :

```
grandpere(x,y) -> pere(x,z) pere(z,y);
```

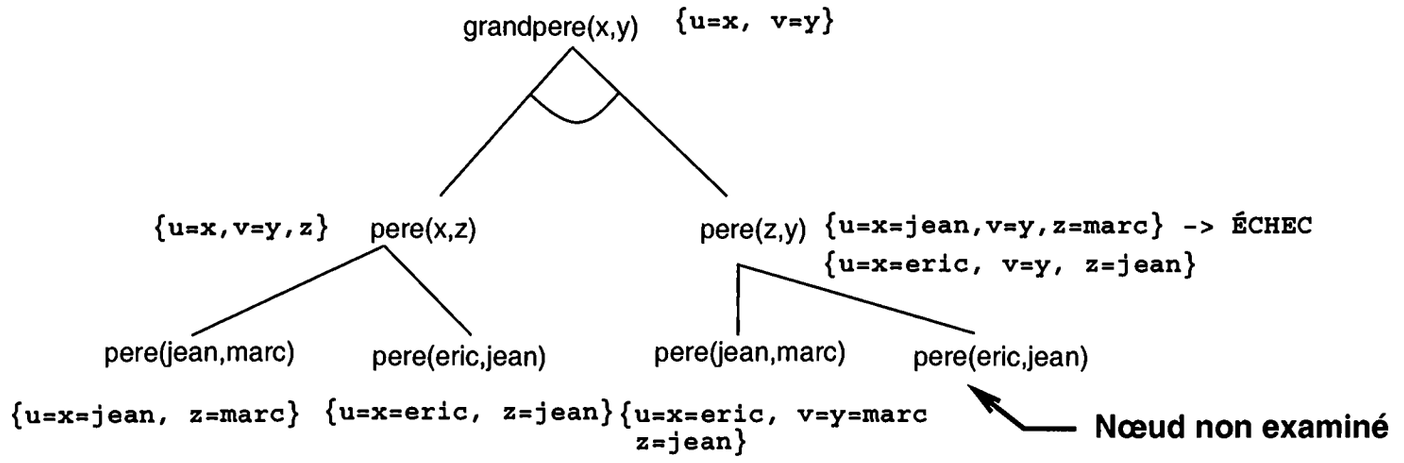


Figure 18.3 – Résolution avec variables et retour-arrière

Il lie la variable u de la question à la variable x de la clause, et la variable v à la variable y.

2. il tente de résoudre le premier sous-but `pere(x,z)`. Il sélectionne la première clause s'unifiant avec `pere(x,z)`, `pere(jean,marc)`, et lie la variable x à `jean` et z à `marc`. Par transitivité, u est lié à `jean` ;
3. le premier sous-but étant résolu, il faut résoudre le sous-but `pere(z,y)` sachant que z est déjà lié à `marc`. Il existe deux possibilités ; la première est `pere(jean,marc)`, mais on ne peut unifier z à `jean`. De même, on ne peut unifier z à `eric` (deuxième possibilité) ;
4. il faut donc réaliser un retour-arrière jusqu'au précédent point de choix. C'est le moment où nous avons choisi la clause `pere(jean,marc)` pour résoudre le sous-but `pere(x,z)`. Nous allons maintenant choisir la clause suivante : `pere(eric,jean)`. Mais il faut *restaurer le contexte*. Il s'agit là non seulement de retrouver la question primitive, mais aussi de « libérer » les variables qui ont été liées depuis. Ainsi x ne doit plus être lié à `jean` et z ne doit plus être lié `marc` ;
5. on réalise les nouvelles unifications : x est lié à `eric` et z à `jean` ;
6. il reste à résoudre le second sous-but, `pere(y,z)` avec les nouvelles unifications. Il suffit de sélectionner `pere(jean,marc)`, ce qui unifie finalement y à `marc` et termine la résolution.

Ainsi, tout retour-arrière impose la restauration des valeurs des variables au moment où le choix a été effectué. Ceci oblige le moteur d'inférence à mémoriser dans un contexte les unifications[4], afin de pouvoir remettre la situation « en l'état » lors du retour-arrière.

Enfin, il est bon de signaler le problème suivant : soit le programme élémentaire `toto(x,f(x))-> ;`. Supposons que je pose la question `toto(x,x)`. Le moteur d'inférence va tomber dans une boucle infinie en tentant d'unifier x et $f(x)$. Il devrait, pour éviter ce type de problème, vérifier que la variable x n'apparaît pas dans le terme $f(x)$ avant de réaliser l'unification. Ce test, appelé *test d'occurrence*, n'est jamais réalisé car

4 Le mécanisme véritable est un peu plus complexe, mais nous ne discutons pas ici des subtilités des mécanismes d'unification et des principaux algorithmes que sont le *structure sharing* et le *structure copying*. Le lecteur intéressé peut se rapporter à la littérature appropriée ((Campbell 1984; Boizumault 1988) par exemple).

il transforme un algorithme efficace (au sens de la complexité) en un algorithme inefficace. L'absence de test d'occurrence détruit la correction théorique de la résolution, déjà mise à mal par la recherche en profondeur. Ce problème d'*occur-check* a été habilement contourné dans PROLOG II qui travaille dans le domaine des termes dits *rationnels* dans lequel une équation telle que $x = f(x)$ admet une solution et est simplement soluble.

18.5 Les méta-opérations

Comme nous l'avons vu, le mécanisme de résolution PROLOG est globalement moins satisfaisant (à l'ordre 0) que l'algorithme de résolution de Davis et Putnam. Pourquoi alors tous les systèmes de programmation logique de grande diffusion utilisent-ils le principe PROLOG ? La raison du succès de PROLOG vient de sa capacité à faire autre chose que de la logique pure, et d'être autre chose qu'un langage déclaratif pur. Ainsi, par l'ordonnancement des clauses dans le programme et par l'ordonnancement des sous-buts à l'intérieur des clauses, le programmeur modifie la façon dont le programme s'exécute, puisque PROLOG choisit toujours les clauses dans l'ordre dans lequel elles se trouvent dans la base, et résout les sous-buts dans l'ordre dans lequel ils sont à l'intérieur de la clause.

Mais le programmeur PROLOG peut aussi agir plus directement sur le contrôle.

18.5.1 Modifier le contrôle : le *cut*

En fait, les moyens de contrôle que nous avons décrits jusqu'ici sont insuffisants. Considérons le problème suivant : je souhaite exprimer que tous les gens que je connais font leur service militaire en France, sauf Jean qui est en Allemagne. J'écris alors :

```
service(jean,allemagne)->;
service(x,france)->;
```

En effet, en exprimant ainsi le problème, je sais que le programme commencera par la première clause (qui est le cas particulier), réussira si `jean` est bien l'argument de `service` et ne passera au cas général que si cette clause échoue. Ainsi, si je pose la question `service(jean,x)`, j'aurai bien comme réponse `x=allemagne`. Malheureusement, ce raisonnement est incorrect pour deux raisons au moins :

1. D'une part, certains PROLOG[5] ne s'arrêtent pas à la première solution. Ainsi dans le cas précédent, après avoir trouvé la première solution `x=allemagne`, PROLOG va exécuter un retour-arrière et chercher d'autres solutions. Il trouvera ainsi `x=france`.
2. Supposons maintenant que je pose la question `service(jean,france)`. Pour `jean`, le système devrait, en toute logique, échouer. Mais en fait, il réussit et affiche `Ok!` En effet, il échoue sur la première clause sélectionnée `service(jean,allemagne)->` ; et effectue alors un retour-arrière, sélectionne la seconde clause et réussit !

5 C'est le cas de PROLOG II en particulier.

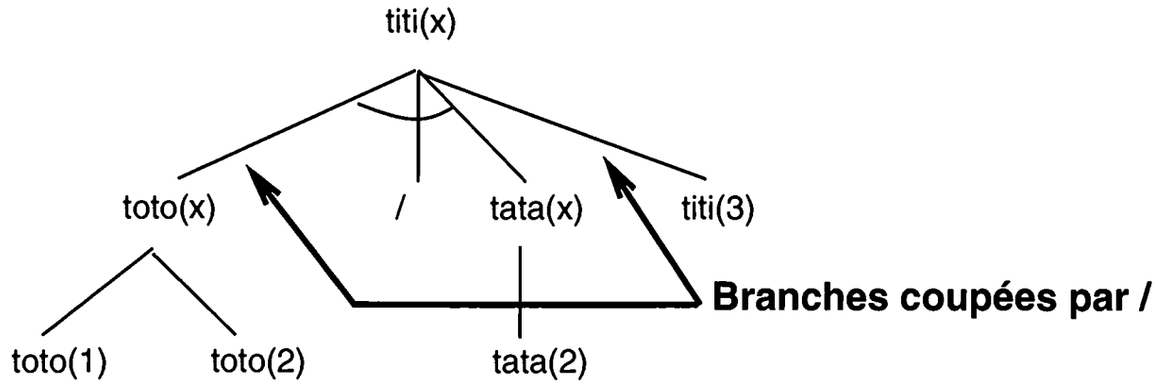


Figure 18.4 – Fonctionnement du *cut*

Il est donc clair que l'on voudrait, dans l'exemple précédent, faire disparaître le point de choix correspondant à la clause `service(jean, allemagne)->;`. Si l'exécution se poursuit avec succès, il ne faut pas que le programme puisse effectuer un retour-arrière à cet endroit là. PROLOG a prévu un semblable mécanisme le *cut*, représenté par un / dans les programmes PROLOG II. Ainsi le programme correct est :

```
service(jean,allemagne) -> /;
service(x,france) ->;
```

Le *cut* supprime tous les points de choix placés entre le début de résolution de la clause et le moment où il est résolu. Le nom de *cut*[6] vient de la représentation d'un programme PROLOG sous forme d'arbre : en effet, le *cut* coupe toutes les branches ET qui sont à sa gauche et qui dépendent du même nœud que lui, et toutes les branches OU au même niveau que lui. Nous allons voir cela sur un exemple un peu plus complexe où le *cut* ne se trouve pas en fin ou en début de clause, mais au milieu :

```
titi(x) -> toto(x) / tata(x);
titi(3) ->;
toto(1) ->;
toto(2) ->;
tata(2) ->;
```

Nous avons représenté sur la figure 18.4 l'exécution de ce programme pour la question `titi(y)`. Reprenons en détail ce mécanisme :

1. Le moteur unifie la question `titi(y)` avec la tête de la clause `titi(x) -> toto(x) / tata(x)` ; (x lié à y).
2. le moteur résout le premier sous-but `toto(x)`. Il y a deux possibilités, il choisit la première `toto(1)` et unifie x à 1.
3. Il exécute le deuxième sous-but, / (*cut*). Le *cut* coupe la branche ET correspondant à `toto(x)` (elle est à sa gauche), et donc *supprime toutes les sous-branches dépendant de* `toto(x)`. Ainsi le sous-choix `toto(2)` est supprimé. Le *cut élimine également les branches OU à sa droite* donc le sous-choix `titi(3)` est également éliminé.

6 Coupure en français.

4. Le moteur résout le troisième sous-but `tata(x)`. Mais x est déjà lié à 1 et donc `tata(2)` ne peut s'unifier.
5. Le moteur effectue un retour-arrière. Mais il ne reste aucun point de choix donc la résolution échoue.

Remarquons que la résolution réussit en revanche, si nous posons les questions `titi(3)` ou `titi(2)`. En fait, la construction que nous avons réalisée au-dessus a une interprétation procédurale simple : elle signifie « pour résoudre `titi(x)` : si `toto(x)` alors `tata(x)` sinon `titi(3)` ». De façon générale, le *cut* est bien souvent un moyen de transformer le langage PROLOG déclaratif en un langage procédural classique. Il est bien souvent utilisé n'importe comment, et certains programmes écrits en PROLOG avec un *cut* par clause sont en fait des programmes procéduraux classiques (déterministes au sens des machines de Turing *déterministes*). Si le *cut* est un puissant moyen de contrôle, c'est aussi une grande tentation offerte au programmeur de détruire la structure déclarative du langage.

18.5.2 Les prédicats prédéfinis

Les symboles de prédicats prédéfinis sont des symboles de prédicats dont la résolution s'effectue différemment. Quand le moteur d'inférence rencontre un symbole de prédicat prédéfini, il exécute une fonction particulière définie à l'avance. Les symboles de prédicats prédéfinis permettent en particulier d'implanter les opérations arithmétiques. Par exemple : `val(add(3,4),x)` unifie x à 7 (3+4).

Ils permettent également d'effectuer des opérations d'entrée-sortie. Ainsi le prédicat `outm(objet)` réussira toujours et affichera la valeur de `objet`. On peut également saisir la valeur d'un caractère ou d'un entier au clavier : le prédicat `in-integer(x)` unifiera à x la valeur d'un entier entré au clavier. Notons que les prédicats pré-définis dépendent totalement de la partie procédurale du langage. C'est parce que l'on connaît l'ordre d'appel des prédicats que l'on peut se permettre de faire lire une variable au clavier. Si les prédicats étaient résolus dans n'importe quel ordre, la lecture d'une variable n'aurait pas grande signification.

18.5.3 L'évaluation gelée

Certains PROLOG, dont PROLOG II présentent la caractéristique de pouvoir geler, ou retarder l'évaluation de certains prédicats prédéfinis.

Voyons ceci sur un exemple. Le prédicat prédéfini `dif(x,y)` permet en PROLOG II de tester si deux variables sont différentes. Dans ce cas, l'évaluation de `dif(x,y)` est un succès ; si les variables sont égales, l'évaluation est un échec. Mais que faire si une variable (ou les deux) est libre au moment de l'évaluation ? On ne peut savoir à ce moment si les deux variables sont ou ne sont pas égales. Face à cette situation, PROLOG II *gèle* l'évaluation du prédicat : il mémorise l'équation $x \neq y$, qui devra toujours être vérifiée. Si une unification venait à se produire et qu'elle violait cette inégalité, l'unification échouerait. En fait, PROLOG II est capable de gérer des contraintes simples pendant son exécution[7].

---

7 Ce mécanisme de PROLOG II sera étendu en PROLOG III pour faire de la programmation logique avec contraintes.

18.5.4 Le problème de la négation

Nous avons vu comment représenter en PROLOG la phrase « tous les gens que je connais font leur service militaire en France, sauf Jean qui le fait en Allemagne » en utilisant le *cut*. Nous allons voir que le *cut* permet également de résoudre partiellement le problème de la négation en PROLOG. Supposons que l'on souhaite, dans un programme, vérifier non pas `toto` mais la négation de `toto`. Je peux écrire le programme suivant, valable en fait pour tout but dont on veut vérifier la négation :

```
not(p) -> p / echec;
not(p)->;
```

Ici le prédicat `echec` est un prédicat quelconque qui doit simplement ne jamais apparaître dans la tête d'une règle, ce qui fait que sa résolution entraîne toujours un échec (d'où son nom). Que se passe-t-il lorsque je pose la question `not(toto)` ? Le système commence par essayer de prouver `toto`. S'il y parvient, le *cut* supprime les choix en attente (l'autre alternative du `not`), et exécute le prédicat `echec`, qui échoue. Donc, si `toto` est résoluble, `not(toto)` échoue. En revanche, si le moteur ne parvient pas à résoudre `toto`, le *cut* n'est pas exécuté et le système passe à la seconde alternative de `not` qui, elle, réussit systématiquement. Ainsi, la négation est implantée en PROLOG.

Remarquons cependant que cette négation dite *négation par échec* n'est pas sans rapport avec l'*hypothèse du monde clos* (tout ce qui n'est pas explicitement vrai est faux). En effet, on suppose que `non(p)` est vrai si on ne parvient pas à prouver p.

18.6 PROLOG et parallélisme

Comparé aux langages compilés traditionnels, PROLOG est par essence un langage lent[8]. En revanche, il possède une qualité fondamentale, bien qu'encore peu exploitée : il est facilement parallélisable.

Le parallélisme est trivial à implanter sur les arcs *OU*. Il s'agit, comme nous l'avions montré au chapitre 15, de distribuer la résolution. Chaque fois que nous rencontrons un arc *OU* dans notre résolution, nous déléguons la sous-tâche correspondant à la résolution sur cet arc à un processeur particulier, en lui communiquant l'état courant de la résolution. Il suffit alors au processeur de continuer la recherche jusqu'à l'obtention d'une solution qu'il communique au processeur maître.

Le parallélisme est plus complexe à implanter sur les arcs *ET*, surtout pour le rendre efficace. En effet, les sous-tâches issues d'un même arc *ET* ne sont pas indépendantes. En particulier, les unifications réalisées par chacune des sous-tâches doivent être communiquées aux autres.

---

8 Il existe des « compilateurs » PROLOG ; il ne faut cependant pas se leurrer : on ne peut totalement compiler PROLOG. Le langage possède des propriétés (la possibilité de faire des `assert` par exemple) qui empêchent une compilation totale (de façon plus élémentaire encore, la question pouvant être quelconque, il faudrait recompiler le programme pour chaque question). La compilation PROLOG n'a pas grand chose à voir avec la compilation traditionnelle de langage comme PASCAL ou C.

Enfin, signalons deux derniers problèmes dans le parallélisme. Si nous nous limitions au PROLOG pur (pas de *cut*, pas de prédicat d'entrée-sortie), tout se passerait parfaitement. En revanche, le *cut* introduit de sérieux problèmes dans le parallélisme sur les arcs *OU*, puisque le fait de rencontrer un *cut* doit annuler la résolution sur tous les arcs qui se trouvent au même niveau. Enfin, les prédicats d'entrée-sortie posent d'importants problèmes pour les arcs *ET* car ils sont sensés être bloquants lors de leur résolution.

Pour une discussion générale sur le parallélisme et PROLOG, on peut lire (Percebois and others 1986) ou (Percebois and others 1987) (en Français) qui présente une implantation d'un PROLOG parallèle ET/OU sur transputers (projet COALA) ou (Santos Costa *et al.* 1991c; 1991b; 1991a) (projet Andorra). Une très bonne étude sur les PROLOG parallèles de type OU et sur les techniques susceptibles de les améliorer, a été faite par Bogumil Hausman dans (Hausman 1989).

Il existe une vaste littérature sur les PROLOG parallèles, mais ces langages n'ont, pour l'instant, pas dépassé le stade de la recherche.

18.7 Programmation logique avec contraintes

L'apparition de la programmation logique avec contraintes résulte naturellement de l'évolution de PROLOG II, et en particulier de son algorithme d'unification qui repose sur la résolution d'un système d'équations portant sur des arbres (représentant les termes PROLOG II). Unifier deux termes t_1 et t_2 consiste alors à satisfaire la contrainte d'égalité des deux arbres représentant $t1$ et $t2$ (contrainte notée $t1 = t2$) en respectant certaines contraintes sur les variables présentes dans $t1$ et $t2$[9].

Cette évolution s'est naturellement poursuivie vers l'introduction de nouveaux domaines plus riches que le domaine de Herbrand, sur lesquels peuvent porter des contraintes plus complexes, et plus proches du monde réel. L'engouement naturel que l'on observe à l'égard de ces langages provient surtout du fait que la programmation logique avec contraintes a un domaine d'application immédiat très vaste, puisque la majorité des problèmes se posant en terme de satisfaction de contraintes dans le domaine choisi pourront s'y modéliser.

Ces problèmes, souvent difficiles (au sens de la complexité), n'ont pas de formalisme d'expression et de méthodes de résolution *effectives* générales. La PLC offre donc un cadre d'expression général, permettant *a priori* un codage rapide et aisé de ces problèmes. De plus, elle permet à l'utilisateur de profiter des algorithmes de résolution de contraintes perfectionnés qui sont inclus dans le langage : certains ont parlé de « recherche opérationnelle déclarative ».

Ces langages ne constituent pas pour autant une réponse au caractère NP-complet de certains de ces problèmes. Seule la formulation du problème peut être rendue, parfois, plus aisée et plus simple à maintenir.

Deux voies sont apparentes à l'heure actuelle.

La première, dont les représentants sont les dialectes PROLOG III (Colmerauer 1986; Pro 1990) et CLP($\mathbb{R}$) (Jaffar *et al.* 1990) s'appuie sur un modèle théorique s'inspirant des

9 « affectation sylvestre »–portant sur des arbres–qui joue le rôle des substitutions.

systèmes de réécriture (PROLOG III), et de la logique (CLP($\mathbb{R}$)). L'essentiel des propriétés théoriques de PROLOG (en particulier sa sémantique du point fixe (Lloyd 1987)) est conservé.

La seconde, dont les représentants sont plus variés (PECOS (ILO 1992), CHARME (BUL 1990), voire CHIP (Int 1990)), sacrifie un peu les propriétés théoriques (et abandonnent éventuellement le langage PROLOG) afin de pouvoir profiter de la puissance d'expression et de résolution des CSP (Constraint Satisfaction Problems, présentés au chapitre 14).

Nous nous restreindrons à l'étude des premiers, en présentant rapidement le schéma CLP($\mathcal{X}$) permettant d'étendre PROLOG à d'autres domaines que le domaine de Herbrand, puis nous présenterons le langage PROLOG III (décrit dans (Colmerauer 1986; Pro 1990)), les relations et domaines introduits.

18.7.1 Le schéma CLP($\mathcal{X}$)

De façon peut-être un peu abusivement synthétique (se reporter aux chapitres 7 et 18 pour plus de détails), on peut en fait résumer les éléments clefs de la programmation logique par :

- *un langage* : celui des clauses de Horn[10] du premier ordre. Une clause est formée de symboles de prédicats appliqués à des termes du premier ordre *i.e.*, des termes obtenus par application de symboles de fonctions à des symboles de constantes et des symboles de variables. Un terme est interprété par défaut dans l'univers de Herbrand (cf. §7.4) ;
- *un système d'inférence* : le principe de résolution. Son application sur des clauses d'ordre un nécessite la résolution du problème d'unification via un algorithme permettant de calculer une solution « la plus générale » possible à un système de contraintes d'égalités entre termes avec variable, donc des contraintes dans l'univers de Herbrand. C'est à partir de ces « solutions les plus générales » que seront construites les réponses données à l'utilisateur.

Il en découle un ensemble de propriétés théoriques fort agréables dont la plus marquante est l'équivalence entre les différentes sémantiques (opérationnelle, par point fixe, logique) d'un programme.

Dès lors que l'on désire étendre le domaine de calcul de PROLOG à d'autres domaines que le domaine de Herbrand, un ensemble de problèmes doivent être résolus :

- il faut étendre le langage pour pouvoir faire référence à des objets du domaine (et donc donner une interprétation des prédicats, fonctions et constantes ajoutées qui ne se fera plus dans l'univers de Herbrand) ;
- il faut étendre la définition de l'unification pour pouvoir prendre en compte les nouveaux termes et prédicats introduits dans le langage. L'algorithme chargé de calculer les « solutions les plus générales » imposera alors la nature des réponses retournées à l'utilisateur.

Une grande variété de domaines pourraient être considérés pour « étendre » PROLOG. Afin de placer l'ensemble de ces langages dans un cadre théorique commun, J. Jaffar et J.L. Lassez ont conçu le schéma de langage CLP($\mathcal{X}$) (CLP pour *Constraint Logic Programming*, la variable $\mathcal{X}$ représentant les différents domaines dans lesquels on peut instancier

10 plus précisément les clauses de Horn définies *i.e.*, ayant *un* atome en partie conséquent.

ce schéma). PROLOG III est ainsi un CLP(arbres rationnels, $\mathbb{R}$, $\mathbb{Q}$, Booléens), CLP($\mathbb{R}$) est un CLP(arbres finis, $\mathbb{R}$).

Considérons un domaine $\mathcal{D}$. Tout comme pour les termes d'ordre un, on va considérer un ensemble Σ de noms de fonction (avec leur arité, une fonction d'arité nulle sera une constante) et un ensemble Π de symboles de prédicats (avec leur arité). À la différence des termes d'ordre un, on donnera *a priori* une interprétation pour ces symboles.

- pour chaque symbole de fonction $f \in \Sigma$ d'arité n, on définit la fonction de $\mathcal{D}^n \rightarrow \mathcal{D}$ qui lui donne un sens ;
- pour chaque symbole de prédicat $p \in \Pi$ d'arité n, on définit la relation de $\mathcal{D}^n \rightarrow \{v, f\}$ qui lui donne un sens.

On peut ainsi envisager comme structure $\mathcal{X}$ =(domaine(s), symboles de fonctions et de prédicats, interprétation des symboles) :

- l'univers de Herbrand, muni des fonctions de formation de termes et de la relation d'égalité (qui correspond à PROLOG) ;
- le domaine des entiers naturels, muni de $(+, \times)$ (avec leur sens habituel) et de la relation $=$;
- le domaine des réels, muni de $+$ et de la multiplication par une constante (unaire), des relations $=, \leq, \geq, <, >$;
- le domaine des réels, muni de $+, \times$, des relations $=, \leq, \geq, <, >$;
- le domaine des booléens, équipé de l'algèbre habituelle $(\vee, \wedge, \neg, \rightarrow, \leftrightarrow \ldots)$ et de la relation d'égalité ;
- tout ensemble fini, sans aucune fonction, et muni des relations $(=, \neq)$, ou d'autres relations...

Une fois la structure définie, et étant donné un ensemble de symboles de variables, il est assez simple d'étendre la définition des termes du premier ordre à une structure $\mathcal{X}$:

Définition 18.1 – $\mathcal{X}$-terme – *Un $\mathcal{X}$-terme est défini récursivement par :*

- *un symbole de variable est un terme ;*
- *si $f \in \Sigma$ est un symbole de fonction d'arité n et si $t_1, \ldots, t_n$ sont des $\mathcal{X}$-termes, alors $f(t_1, t_2, \ldots, t_n)$ est un $\mathcal{X}$-terme.*

On définit alors une $\mathcal{X}$-contrainte :

Définition 18.2 – $\mathcal{X}$-contrainte – *Une $\mathcal{X}$-contrainte est définie récursivement par :*

- *si $p \in \Pi$ est un symbole de prédicat d'arité n et si $t_1, \ldots, t_n$ sont des $\mathcal{X}$-termes, alors $p(t_1, t_2, \ldots, t_n)$ est une $\mathcal{X}$-contrainte (dite primitive) ;*
- *si $p_1, \ldots, p_n$ sont des $\mathcal{X}$-contraintes, alors la conjonction $\{p_1, \ldots, p_n\}$ est une $\mathcal{X}$-contrainte.*

Pour la structure $\mathcal{X} = (\mathbb{N}, \{+, -, \times\}, \{=\})$:

- $x + y, 3 \times x + 2 \times y + 4, 3 \times x \times x + y$ sont des exemples de $\mathcal{X}$-termes ;
- $\{x + y = 3 \times x + 2 \times y + 4, 3 \times x \times x + y = 2\}$ est une $\mathcal{X}$-contrainte formée par la conjonction de deux $\mathcal{X}$-contraintes primitives ;

On dira qu'une $\mathcal{X}$-contrainte est satisfiable s'il existe une affectation des variables de la contrainte qui la rend vraie dans l'interprétation donnée par la structure $\mathcal{X}$.

Un programme de CLP($\mathcal{X}$) est alors formé d'une conjonction de $\mathcal{X}$-clauses de Horn définies, de la forme :

$$H :- c, B_1, \ldots, B_n$$

dans laquelle H et les B_i sont les atomes habituels formés à partir de symboles de prédicats (non interprétés) appliqués à des $\mathcal{X}$-termes et c une $\mathcal{X}$-contrainte.

Pour la même structure $\mathcal{X} = (\mathbb{N}, \{+, -, \times\}, \{=\})$ que précédemment, on peut considérer le programme suivant, définissant la suite de Leonardo Fibonacci :

```
fib(0, 1).
fib(1, 1).
fib(X, N + M) :-  X > 1,
                  fib(X - 1,N),
                  fib(X - 2,M).
```

Un apport essentiel des contraintes est la réversibilité complète du programme ainsi défini qui peut calculer la valeur de $fib(n)$ mais aussi, si il existe, le n pour lequel $fib(n)$ prend une valeur donnée.

On étend de la même façon la notion de but PROLOG : il est possible d'utiliser une conjonction d'une $\mathcal{X}$-contrainte et d'atomes. Le mécanisme de résolution est alors simplement étendu. Étant donnés un programme $\mathcal{P}$ et $k, \{Q_i\}$ un but. Soit $H :- c, \{B_i\}$ une clause de $\mathcal{P}$, il est possible de se ramener, en un pas de dérivation, au but :

$$k \cup c \cup \{Q_1 = H\}, \{Q_{i \geq 2}\} \cup \{B_i\}$$

ssi la $\mathcal{X}$-contrainte $k \cup c \cup \{Q_1 = H\}$ est satisfiable.

Une séquence de dérivations réussit si elle s'achève sur un but formé uniquement d'un ensemble de contraintes (qui est satisfiable). Cet ensemble forme une réponse au but. La stratégie choisit pour effectuer ces dérivations est habituellement la même qu'en PROLOG : les clauses du programme sont examinées en séquence, les atomes (non interprétés) du but traités de gauche à droite.

Jaffar et Lassez (Jaffar *et al.* 1985; Jaffar and Lassez 1987) ont montré que les propriétés fondamentales de PROLOG s'étendent à CLP($\mathcal{X}$)[11] pourvu que la détermination de la satisfiabilité d'une $\mathcal{X}$-contrainte soit complète (une $\mathcal{X}$-contrainte est soit prouvée satisfiable, soit prouvée insatisfiable, certaines relaxations peuvent être apportées à cette propriété) et pourvu que la structure $\mathcal{X}$ soit *solution-compacte i.e.* :

1. chaque élément du domaine $\mathcal{D}$ peut être défini par un ensemble (éventuellement infini) de contraintes ;
2. le complément d'une contrainte (*i.e.*, l'ensemble des éléments de $\mathcal{D}$ ne satisfaisant pas la contrainte) peut être défini par un ensemble (éventuellement infini) de contraintes.

Parmi les structures précédemment considérées :

- le domaine des arbres rationnels, muni des fonctions de formation de termes et des relations d'égalité et de différence (qui correspond à PROLOG II) est complet pour la satisfaction (algorithme d'unification de PROLOG II) et solution-compact ;
- le domaine des entiers naturels, muni de $(+, \times)$ et de la relation $=$ n'est pas complet pour la satisfaction (la résolution d'une équation diophantienne est un problème indécidable, cf. chapitre 10.) ;

11 Complétude d'une implantation correcte, existence d'une sémantique du plus petit point fixe, équivalence des différentes sémantiques : opérationnelle, par point fixe, logique (en considérant une théorie τ pour le domaine et les opérations sur ce domaine) et algébrique,...

- le domaine des réels, muni de + et de la multiplication par une constante (unaire), des relations $=, \leq, \geq, <, >$ est complet pour la satisfaction (par une généralisation de l'algorithme du simplexe par exemple) et solution-compact ;
- l'ensemble des réels, muni des relations $(=, <, >, \leq, \geq)$ et des opérations $(+, *)$ forme une structure solution-compacte, complète pour la satisfaction (procédure de décision de Tarski) ;
- tout ensemble fini est complet pour la satisfaction (procédure de *generate and test* par exemple, tout autre algorithme CSP conviendra), ceci pour toute relation. Il définira une structure solution-compacte si les relations $=, \neq$ sont considérées ;
- le domaine des booléens en est un cas particulier, complet pour la satisfaction (algorithme de Quine, procédure de Davis & Putnam, champ de production (Siegel 1987), évaluation sémantique (Oxusoff and Rauzy 1989), P-clash (Palmade 1992), Ordered Binary Decision diagrams (Bryant 1986; Brace *et al.* 1990)...) et trivialement solution compact.

18.7.2 Le langage PROLOG III

Rappelons tout d'abord qu'il a été démontré que PROLOG II est une instance de CLP($\mathcal{X}$) (Jaffar *et al.* 1986). Tout comme PROLOG II, le langage PROLOG III manipule exclusivement des *arbres* dont chaque nœud est étiqueté, et qui peuvent être éventuellement réduits à une feuille (dont une représentation syntaxique pourra être donnée par des termes).

L'apport essentiel consiste en la possibilité d'exprimer des *relations* qui traduisent des conditions portant sur les arbres (<aa,bb,cc> $\neq$ <11,22,33>, $3 > 2$...). Une relation appliquée à un n-uplet d'arbres est, ou n'est pas, vérifiée. Il ne s'agit cependant pas d'une opération à valeurs dans les booléens de PROLOG III.

Les relations disponibles sont :
- égalité et inégalité sur les arbres, notées = et # ;
- l'implication entre booléens (notée comme l'opération d'implication =>) ;
- les relations d'ordre sur les nombres (rationnels ou flottants), notées >, <, >=, <=.
- les relations unaires de typage et de longueur de n-uplets (`!num`, `!bool`...).

Une *contrainte* porte sur un ou plusieurs termes. C'est l'expression syntaxique d'une relation sur les éléments des ensembles d'arbres représentés par les termes. Un ensemble (ou système) de contraintes est soluble (satisfiable) s'il existe une affectation des variables apparaissant dans les termes de la contrainte qui permet de vérifier la relation exprimée.

La base de connaissances d'un *programme* PROLOG III est formée d'un ensemble de *règles* dont la forme résulte d'une évolution des clauses de Horn du modèle initial :

$$t_0 \text{ -> } t_1 \ldots t_n \ , \ \mathrm{S}$$

où

$t_0 \ldots t_n$ sont des termes PROLOG III.
S est un ensemble fini (ou système) de contraintes.

Sémantiquement et informellement, une règle s'interprète par :

Si S est satisfait et si $t_1 \ldots t_n$ sont vrais, alors t_0 est vrai

La méthode utilisée permet de produire, à partir d'un but :

$$C_1 \ldots\ C_m\ ,\ S_i$$

et d'une clause de la base de connaissances :

$$B_1\ \text{->}\ B_2 \ldots\ B_n\ ,\ S_0$$

le nouveau but :

$$C_2 \ldots C_m\ B_2 \ldots B_n\ ,\ S_i \cup S_0 \cup \{B_1 = C_1\}$$

dont le système de contraintes associé ($S_i \cup S_0 \cup \{B_1 = C_1\}$) doit être satisfiable.

La stratégie utilisée est la même qu'en PROLOG et en PROLOG II, elle consiste à explorer un arbre dont chaque nœud est un but et dont chaque fils résulte de l'application de la méthode précédente. Les feuilles de l'arbre correspondent aux échecs (système de contraintes devenu insatisfiable) et aux succès (but égal à la clause vide avec un système de contraintes satisfiable). L'arrivée à une feuille de succès entraîne l'affichage du système de contraintes sous une forme « simplifiée » et dans les termes des variables qui sont dans le but seulement (projection de l'ensemble solution sur ces variables). Ses propriétés sont les mêmes que celles de la résolution PROLOG classique.

Notons tout de même qu'il est nécessaire de pouvoir vérifier la satisfiabilité du système de contraintes très rapidement puisqu'elle doit être établie à chaque inférence. Il est possible de profiter de l'évolution incrémentale du système de contraintes ($S_{i+1} = S_i + \cdots$) pour améliorer l'efficacité de résolution. C'est ainsi que les différentes versions du simplexe utilisées par PROLOG III sont des versions incrémentales de l'algorithme original. Il s'agit d'une limitation importante des langages de PLC purs tels que PROLOG III ou CLP($\mathbb{R}$) qui ne peuvent pas s'appuyer sur des algorithmes de résolution dans les domaines discrets, car les plus efficaces (Backtrack+Arc-consistance par exemple) ne sont pas incrémentaux.

D'autre part une certaine latitude apparaît dans la nature des réponses données à l'utilisateur : il s'agira d'un système de contraintes « simplifié ». Jusqu'à quel point et dans quelle direction ?

Dans l'éventualité, où l'on décide d'utiliser PROLOG III pour travailler sur les entiers, le prédicat `enum` permet d'énumérer l'ensemble des valeurs entières qui respectent les contraintes courantes portant sur une variable. Ainsi, le but :

$$\texttt{enum(X)}\{2X > 9, 3x < 28\}$$

fournira-t-il les réponses $\{X = 5\}, \{X = 6\}, \{X = 7\}, \{X = 8\}\{X = 9\}$?

On est ramené dans ce cas à un algorithme de type *Backtrack* informé[12], où `enum` permet d'instancier les variables en séquence. PROLOG III ne disposant d'aucune heuristique pour l'ordre d'instanciation des variables, le temps de résolution est souvent prohibitif (en comparaison avec ce que permettent d'obtenir, par exemple, les algorithmes de choix et propagation issus des techniques CSP présentées au chapitre 14). La nature des réponses change aussi puisqu'une réponse sera formée d'une (ou d'un ensemble de) affectation rendant vrai le but et non plus d'un système de contraintes « simplifié ».

---

12 Assez proche finalement des algorithmes hybrides des CSP, puisqu'après chaque instanciation, le système de contraintes courant est « simplifié ».

18.7.3 Conclusion

La réalisation effective d'un langage du type CLP($\mathcal{X}$) doit être le résultat d'un compromis des plus délicats : il faut faire le choix d'un domaine, de relations et de fonctions qui soient pragmatiquement intéressants (il est simple d'y représenter des problèmes réels), mais pour lesquels les procédures de résolution efficaces existent. On observe, dans la majorité des cas, que les problèmes intéressants sont naturellement les plus difficiles (du point de vue de la complexité temporelle). On peut donc :

- se limiter à un domaine et un langage de contraintes pour lesquels tout problème de satisfaction est polynomial. Le choix le plus souvent réalisé (PROLOG III, CLP($\mathbb{R}$), CHIP) consiste à prendre comme domaine les rationnels ou les réels (approchés par les flottants, ce qui peut poser de gros problèmes d'exactitude) et comme seules contraintes exprimables les contraintes linéaires (définissant un problème polynomial). Tout le caractère NP-complet d'un problème se retrouvera alors, une fois codé, dans la partie PROLOG. Ainsi, le problème d'ordonnancement avec ressources[13] (cf. chapitre 11) fait naturellement apparaître des contraintes dites *disjonctives* entre les dates de démarrage S_A et S_B de deux tâches A et B de durées d_A et d_B nécessitant une même ressource non partageable :

$$(S_A \geq S_B + d_B) \quad \vee \quad (S_B \geq S_A + d_A)$$

  Informellement : A commence après B ou B commence après A. Le codage de cette contrainte peut se faire (par exemple en PROLOG III) via deux clauses, séparant bien le caractère exponentiel du problème :

```
avant-apres(SA,SB,DA,DB) -> {SA >= SB+DB};
avant-apres(SA,SB,DA,DB) -> {SB >= SA+DA};
```

  Chaque contrainte disjonctive entraîne la création d'un point de choix supplémentaire dans l'arbre de recherche. Dans le pire des cas, chacun de ces points de choix devra être considéré et n contraintes disjonctives entraîneront la résolution de 2^n systèmes de contraintes ! Le programme obtenu est résolu de façon relativement inefficace par PROLOG III, ou tout autre langage ayant fait le choix d'une structure similaire[14].
- choisir un domaine et un langage de contraintes pour lesquels le problème de satisfaction est difficile. Les instances les plus souvent rencontrées sont celles utilisant :
  - **l'algèbre de Boole** (PROLOG III, CHIP) : on dispose d'un langage bien défini et les algorithmes ne manquent pas. Le choix que peut faire le concepteur du langage est surtout au niveau du type de réponses fournies à l'utilisateur : un ensemble de formules équivalent à l'ensemble des contraintes accumulées et minimal (dans un sens qu'il faut définir) ou simplement une séquence de modèles de cet ensemble. Le premier choix est extrêmement coûteux en temps de calcul, mais c'est celui vers lequel la majorité des langages ont convergé dans ce cas précis.

---

13 Le même type de contraintes se retrouve dans les problèmes de raisonnement géométrique, à des ordres supérieurs.

14 P. van Hentenryck a proposé l'introduction d'un opérateur, dit de cardinalité, pour résoudre partiellement ce problème (van Hentenryck and Deville 1991). Il faut noter que des propositions similaires ont été faites dans le cadre de la logique propositionnelle (Hooker 1988; Benhamou and Sais 1992), avec des résultats plus intéressants, nous semble-t-il.

- **les domaines finis** : il peut s'agir d'ensembles de symboles munis des simples relations d'égalité et de différence ou de structures plus riches (intervalles d'entiers, munis de $+, -, *, /$ ... et des relations habituelles). Les algorithmes CSP (cf. chapitre 14) sont ainsi utilisés dans les langages CHIP, CHARME, PECOS. Les algorithmes peuvent être souvent spécialisés pour tenir compte des spécificités des structures choisies. La mauvaise adéquation des algorithmes CSP au cadre CLP($\mathcal{X}$) a amené les concepteurs à affaiblir la propriété de satisfiabilité (qu'il faudrait vérifier à chaque dérivation) en une propriété de consistance locale (le plus souvent l'arc-consistance). Les réponses ne définissent plus alors l'ensemble des solutions, mais un sur-ensemble de l'ensemble des solutions (les domaines des variables sont réduits, mais il se peut que des valeurs ne participant à aucune solution soient données en réponse). L'utilisateur peut sinon demander une satisfaction complète (réalisés le plus souvent via un algorithme de type **forward checking**, cf. chapitre 14). Une réponse sera alors formée d'une (ou d'un ensemble de) affectation rendant vrai le but et non d'un système de contraintes « simplifié ».

Le grand nombre de langages réalisés et d'utilisations pratiques observées pourraient amener à penser que la seconde approche est la plus intéressante, mais les domaines d'applications sont assez distincts. Il est certain que la généralité du cadre CSP rend bien souvent la modélisation des problèmes plus simples (dès lors que le langage profite de cette richesse, ce qui n'est pas toujours le cas).

Partie V

Apprentissage

Un ingénieur, un enseignant et un chercheur en Intelligence Artificielle cognitive discutaient afin de déterminer quelle était la plus grande réalisation de l'humanité. L'ingénieur prétendait qu'il s'agissait de la roue. L'enseignant quant à lui pensait que c'était certainement la presse d'imprimerie. Finalement, le chercheur en IA déclara que cela ne pouvait être que le thermos. Ses deux collègues en furent stupéfiés. « Mais qu'a donc le thermos de si fabuleux ? » demanda l'ingénieur. « Et bien, » répondit le chercheur en IA, « si on y met des choses chaudes, il les garde chaudes, alors que si on y met des choses froides, il les garde froides. » « Et alors ? » demanda l'enseignant. « Alors ? ! », répondit le chercheur en IA, visiblement surpris, « mais comment fait-il donc pour savoir ? »

— *Anonyme*

CHAPITRE 19

Apprentissage Symbolique Automatique

Jean-Marc Alliot

19.1 Présentation

Être en mesure de faire apprendre des connaissances aux machines est rapidement apparu nécessaire à la plupart des chercheurs, pour de nombreuses raisons. Sur un plan « philosophique », les gens se sont rapidement aperçus de l'importance fondamentale de l'apprentissage dans l'intelligence humaine. Prétendre obtenir des systèmes intelligents alors qu'ils sont programmés de façon « ad hoc » pour accomplir une et une seule tâche est une position difficilement tenable. Sur un plan plus pratique, les développements des systèmes experts ont montré qu'un des problèmes principaux, si ce n'est le problème principal, était bien l'acquisition des connaissances ; le trio composé de l'expert, du cogniticien et de l'informaticien a montré quelques-unes de ses limites : d'une part, l'expert a beaucoup de mal à formuler son « savoir » en termes facilement exploitables. D'autre part, la mise en forme de ce savoir par l'informaticien est difficile ; le savoir est, de plus, évolutif et demande de nombreuses révisions des bases de règles qui doivent être effectuées par l'ensemble de l'équipe, et la maintenance de semblables programmes devient rapidement difficile, face à la quantité de connaissances accumulées.

En IA, les premiers travaux sur l'apprentissage remontent probablement aux travaux sur les perceptrons de Rosenblatt (prémisses de l'approche connexionniste que nous discuterons en détail dans le prochain chapitre), et au programme de jeu de dames de Samuel (Samuel 1959) dont nous avons déjà parlé dans le chapitre 15. Cependant, les travaux de Samuel, ainsi que les travaux de ceux qui ont emprunté la même voie que lui (le programme d'Othello BILL, ou même DEEP THOUGHT), ne portent pas sur un apprentissage symbolique du domaine, mais plutôt sur des méthodes numériques. Il s'agit en effet d'optimiser (« d'apprendre ») les coefficients d'une fonction de pondération, mais les notions symboliques fondamentales sont les structures de la fonction de pondération, et elles ont été directement codées par le programmeur. Ainsi, aux échecs, le choix des termes de la fonction de pondération est fondamental (roque, paire de Fous, pions passés, etc.). Le programme ne réalise, suivant l'expression de Feng Hsu, qu'une optimisation des coefficients de la fonction, basée sur une technique des moindres carrés.

Le but de l'apprentissage symbolique est bien différent : selon Yves Kodratoff[1] l'*Apprentissage Symbolique Automatique* (ASA) est « orienté vers l'acquisition de concepts et de connaissances structurés ». Patrick Winston, dans un texte resté célèbre (Winston 1984), a décrit avec un certain enthousiasme les résultats d'un de ses premiers programmes :

> « Confronté à un exemple, le programme construit une représentation structurelle de la scène sous la forme d'un réseau sémantique où les nœuds représentent les composants et les lignes, les relations entre les composants. Après qu'on lui ait dit qu'il s'agissait d'une arche, le programme enregistre cette description comme étant une arche. »

L'ASA fait appel à de nombreuses techniques d'IA que nous avons déjà présentées : la programmation logique, la représentation des connaissances, le maintien de consistance des bases de connaissances (Truth Maintenance System), etc. Elle fait également appel à des techniques mathématiques et des techniques d'informatique théorique. D'autre part, à l'intérieur de l'ASA, coexistent des méthodes différentes. Tout ceci fait de l'ASA un des domaines les plus vastes et les plus complexes à appréhender de l'IA moderne.

Le but général de l'ASA est l'apprentissage à partir d'exemples. L'expert reste dans son cadre de travail puisqu'il doit simplement exercer son expertise ; on ne lui demande en aucun cas de donner des règles, qui seraient déjà une interprétation de son activité.

19.2 Différentes formes d'apprentissage

L'ensemble des travaux sur l'ASA (en Anglais *Artificial Intelligence Approach to Machine Learning*) a commencé à se structurer lors du premier « Machine Learning Workshop » en 1980. Le terme ASA recouvre différentes techniques visant à atteindre des buts parfois différents. Nous ne parlerons pas des méthodes à caractère numérique ; en revanche, nous allons faire un petit lexique[2] des différentes caractéristiques des méthodes symboliques :

EBL/SBL : EBL signifie *Explanation Based Learning* ou *apprentissage par recherche d'explications*. Le système recherche une solution à un problème, et aboutit soit à un succès, soit à un échec. Le programme analyse alors les raisons de l'échec ou du succès et tente d'utiliser ces « explications » pour améliorer le système de résolution.

SBL signifie *Similarity Based Learning* ou *apprentissage par détection de similarités*. En SBL, l'apprentissage se fait à partir d'exemples et de contre-exemples qui sont des échantillons significatifs de la connaissance que l'on souhaite représenter.

Apprentissage Empirique/Rationnel : l'apprentissage est dit *empirique* si le système ajoute « sans se poser de questions » une règle dans sa base de règles, pourvu qu'elle

---

1 Nous avons principalement utilisé trois ouvrages pour la réalisation de ce chapitre : les actes du premier « Machine Learning Workshop » tenu en 1980 à Carnegie-Mellon (Car 1980), le rapport (Cannat *et al.* 1988), et le livre d'Yves Kodratoff (Kodratoff 1986). Le lecteur intéressé par l'Apprentissage Symbolique Automatique pourra se reporter à cet ouvrage, écrit par un des spécialistes mondiaux de l'ASA, pour plus de précisions.

2 Extrait de (Kodratoff 1986).

se soit révélée efficace pour résoudre un exemple ou éliminer un contre-exemple, et que le système reste consistant après son ajout.

En revanche, l'apprentissage est dit *rationnel* si toute adjonction d'une règle dans la base est accompagnée d'une analyse des relations entre cette règle et les règles déjà présentes dans la base.

Apprentissage déductif/inductif/inventif : on dit que l'apprentissage est *déductif* quand le système possède toutes les connaissances nécessaires à la résolution dès le début, *inductif* lorsqu'il est capable de modifier et de faire évoluer certaines représentations internes, *inventif* lorsqu'il est capable de découvrir par lui-même de nouveaux concepts. Comme le reconnaissent les spécialistes eux-mêmes, ce dernier type d'apprentissage automatique n'en est qu'à ses balbutiements.

19.3 EBL

Dans un système d'apprentissage par recherche d'explication (*Explanation Based Learning*), on fournit au système un certain nombre de règles avant l'apprentissage. Ces règles appartiennent à deux catégories :

Les opérateurs initiaux : ce sont les opérateurs de base qui permettent d'effectuer un changement d'état.

Les heuristiques initiales : ces heuristiques indiquent au système comment appliquer les opérateurs initiaux pour produire les solutions.

Les opérateurs initiaux et les heuristiques initiales sont sous-optimaux et le but de l'apprentissage est précisément de trouver de nouveaux opérateurs plus efficaces et de découvrir de nouvelles heuristiques.

Cet apprentissage est connu sous le nom d'apprentissage par recherche d'explication, car c'est en tentant d'identifier les opérateurs responsables du succès ou de l'échec d'une résolution qu'il apprendra ; il est également connu sous le nom d'*apprentissage par l'action* car le mécanisme d'apprentissage est basé sur la génération de solutions (actions) puis leur évaluation. Lors de la définition d'un problème que l'on souhaite résoudre par l'EBL, on doit donc nécessairement inclure dans le système :

- un mécanisme d'évaluation des solutions générées ;
- un mécanisme d'identification de ou des opérateurs responsables du succès ou de l'échec de la résolution (mécanisme d'attribution du mérite) ;
- un mécanisme permettant de modifier les opérateurs ou les heuristiques en fonction des résultats obtenus, de l'évaluation de ces résultats et de l'attribution du mérite.

Le fonctionnement du système s'effectue donc en trois étapes :

1. On fournit au système un exemple de résolution utilisant les opérateurs initiaux (ou on lui laisse construire cette solution s'il en est capable).
2. Le système va tenter de retrouver cette solution lui-même à l'aide des heuristiques initiales. Les heuristiques déclenchent les opérateurs dans un certain ordre suivant en cela des conditions de déclenchement.
3. On modifie les conditions de déclenchement en comparant la solution construite avec la solution modèle. On renforce les conditions de déclenchement améliorant la résolution et on pénalise celles qui la dégradent.

Il existe diverses façons d'appliquer l'EBL. Le lecteur intéressé peut se reporter à (Mitchell 1983) pour la technique dite d'*espace des versions*. Une autre méthode, l'apprentissage par *essais et erreurs* est décrite dans (Langley 1983).

19.4 SBL

Dans l'apprentissage par détection de similarités (*Similarity Based Learning*), l'apprentissage est réalisé par la détection des similarités et des dissimilarités dans un jeu d'exemples et de contre-exemples qui illustrent tous un même concept. On utilise alors un mécanisme de généralisation qui construit un *généralisé*, qui est la partie *apprise*. L'apprentissage par détection de similarité est un processus essentiellement empirique[3] : la construction du généralisé est toujours faite de façon plus ou moins empirique, par opposition à l'EBL. D'autre part, l'apprentissage par détection de similarité doit généralement se faire avec l'aide d'un expert d'un domaine.

19.4.1 Les éléments de base

Les quatre éléments fondamentaux d'un système de SBL sont :

Les descripteurs : il s'agit d'un ensemble de mots qui représentent les propriétés relatives au problème que l'on souhaite traiter.

Les concepts : les concepts sont les éléments du domaine qui apparaissent comme fondamentaux à l'expert du domaine.

Exemples et contre-exemples : un *exemple* de concept est la réunion des descripteurs qui représentent une situation du problème traité illustrant, d'après l'expert, le concept.

La théorie du domaine : il s'agit d'un ensemble de connaissances générales sur le domaine, exprimant des relations entre les descripteurs. Elles sont généralement exprimées sous une forme *si... alors*, et/ou sous forme de taxonomies.

19.4.2 La généralisation

La généralisation est effectuée par le système d'inférence inductive automatique. Il s'agit de déterminer l'ensemble des propriétés qui sont communes à tous les exemples d'un concept et ne sont vérifiées par aucun des contre-exemples ; le résultat[4] est un ensemble de propriétés appelé le *généralisé*. Lors du processus de généralisation, le mécanisme d'induction s'appuie sur la *théorie du domaine* telle qu'elle a été formalisée.

Un généralisé peut être considéré comme une fonction de reconnaissance de concept. On dit qu'une fonction de reconnaissance est *complète* quand elle recouvre l'ensemble des exemples présentés, et qu'elle est *consistante* si elle rejette l'ensemble des contre-exemples.

---

3 Même le SBL dit « rationnel » garde une part d'empirisme.

4 Il n'est pas toujours possible de construire un tel ensemble de propriétés, et ce, pour plusieurs raisons : exemples ou contre-exemples incorrects, mauvaise formalisation du domaine, etc. C'est d'ailleurs là un des problèmes du SBL.

Comme le généralisé est obtenu, en SBL, par inférence inductive sur un ensemble d'exemples et de contre-exemples, on est certain que la fonction de reconnaissance est *complète* et *consistante* pour la description du concept que l'on a. Le but de l'apprentissage est d'obtenir des fonctions de reconnaissance qui soient complètes et consistantes pour l'ensemble du domaine.

La fonction de reconnaissance répond rarement dès la première itération aux attentes placées en elle. Bien souvent, lorsque l'expert examine le généralisé et la fonction de reconnaissance associée, il est capable de trouver des exemples que la fonction de reconnaissance n'accepte pas et/ou des contre-exemples qu'elle accepte. Il faut alors construire un nouveau généralisé prenant en compte ces nouveaux exemples et contre-exemples jusqu'à convergence du processus[5].

19.5 Conclusion

L'apprentissage automatique est une approche originale qui essaie de régler certains problèmes difficiles de l'IA. Actuellement les aspects numériques semblent prendre une place croissante et la tendance est de développer des approches mixtes, associant apprentissage symbolique et techniques d'apprentissage « numériques » du type réseaux de neurones par exemple.

5 Ce mécanisme ne fonctionne, hélas, pas toujours aussi bien. En effet, il suppose que la représentation du domaine (choix des concepts, modélisation de la théorie du domaine) est bien adaptée. Si la modélisation est mal adaptée, on risque, en traitant de nouveaux exemples, de faire apparaître des inconsistances, et en traitant de nouveaux contre-exemples, de faire disparaître la complétude. Il est clair, dans ce cas, que le processus peut ne pas converger.

CHAPITRE 20

Les réseaux de neurones

Jean-Marc Alliot

20.1 L'argument physiologique

L'idée du réseau de neurones formels vient de l'étude du cerveau humain. Ce qui suit n'est qu'une vision très schématique du fonctionnement du dit cerveau[1].

Le cerveau humain est composé d'un ensemble de cellules appelées neurones. Un neurone est composé d'un noyau, de connexions entrantes (les dendrites) et d'une connexion sortante, l'axone. L'influx nerveux se déplace toujours des dendrites vers le noyau, et du noyau vers l'axone. L'influx transmis dans l'axone est fonction de la valeur de l'influx dans chacune des dendrites. Certaines dendrites peuvent avoir un effet moteur favorisant la transmission d'une information dans l'axone, d'autres au contraire ont un effet inhibiteur qui bloque la transmission de l'influx dans l'axone. Il semble que le noyau agisse comme un sommateur des influx venant des dendrites, en affectant un poids (qui peut être négatif) à chacune d'entre elles. Si la somme des influx est supérieure à un seuil, un influx est transmis dans l'axone. Si la somme est inférieure à ce seuil, aucun signal n'est transmis. Un axone peut, par la suite, soit se subdiviser en plusieurs filaments qui serviront chacun d'entrée à d'autres neurones en se connectant aux dendrites de ces neurones via des synapses, soit attaquer directement un élément moteur (muscle par exemple). McCulloch et Pitts sont les « pères » du modèle mathématique du neurone (McCulloch and Pitts 1943). On peut le résumer de la façon suivante ; on pose pour un neurone :

$I_i \in \mathbb{R}$: entrées du neurone
$W_i \in \mathbb{R}$: poids correspondant à chacune des entrées
θ : seuil
$O \in \mathbb{R}$: sortie du neurone
$f(x)$: fonction de seuil vérifiant $\forall x, x > \theta, f(x) = 1$ et $f(x) = 0$ sinon.

Alors

$$O = f(\sum_i W_i I_i)$$

Nous allons dans ce chapitre nous intéresser à différentes techniques mettant en œuvre des réseaux de neurones dans le sens large du terme. Nous allons nous apercevoir qu'il existe de nombreuses approches différentes, ayant des buts différents. Les gens

1 On peut se reporter à l'excellent recueil d'articles (pour la science 1987) pour plus de précisions.

intéressés par le domaine peuvent se reporter, par exemple, à (Hertz *et al.* 1991) (remarquable) ou (Hecht-Nielsen 1990) pour une vision d'ensemble des techniques basées sur les réseaux de neurones, ou en français (Davalo and Naim 1986) qui est un peu plus ancien, ou l'excellent (Bourret *et al.* 1991).

20.2 Les mémoires associatives et le modèle de Hopfield

Dans une mémoire associative, nous stockons dans un réseau de neurones un certain nombre de *formes* et, en fournissant au réseau une partie d'une des formes stockées, nous souhaitons que le réseau reconstitue l'intégralité de la forme stockée.

Un exemple typique d'utilisation d'une mémoire associative est la reconnaissance et la reconstruction d'images. On stocke dans le réseau un certain nombre d'images. Le réseau doit par la suite reconstruire à partir d'un détail qu'on lui fournit l'ensemble de l'image la plus proche de ce détail.

Nous allons détailler dans cette section la façon dont fonctionne une mémoire associative.

20.2.1 Modèle mathématique

Nous allons considérer un réseau comportant N neurones. Tous les neurones sont connectés les uns aux autres[2]. Chacun des neurones pourra fournir une valeur d'activation $+1$ ou -1. La valeur d'activation de l'unité i sera donnée par la formule :

$$O_j = S(\sum_i W_{ji} O_i) \tag{20.1}$$

où les W_{ji} sont les poids du réseau et où S est la fonction « Signe » définie par :

$$S(x) = 1 \text{ si } x \geq 0, \text{ et } S(x) = -1 \text{ si } x < 0$$

Supposons maintenant que nous souhaitions faire évoluer notre réseau. Il est possible de le faire de façon *synchrone* ou de facon *asynchrone*. Dans le mode synchrone, tous les éléments du réseau évoluent en même temps en appliquant simultanément l'équation 20.1. Dans le mode asynchrone, les unités évoluent les unes après les autres, de façon aléatoire par exemple. Dans le cas des mémoires associatives, il est de coutume d'utiliser le mode asynchrone de préférence au mode synchrone.

20.2.2 Mémorisation d'un vecteur binaire

Supposons que nous voulons que notre réseau mémorise le vecteur binaire $\vec{O}^1 = (O_1^1, O_2^1, \ldots, O_n^1)$. Ceci signifie que toutes les valeurs O_i^1 doivent être stables par application de la transformation 20.1 ; donc, il faut :

$$\forall j,\ S(\sum_i W_{ji} O_i^1) = O_j^1$$

---

2 Le modèle élémentaire de McCulloch et Pitts est légèrement modifié. On remplace la fonction de seuil f par la fonction « Signe ». D'autre part, les entrées d'un neurone étant les sorties des autres neurones, on remplace les I_i par des O_i.

Ceci est clairement le cas si nous prenons :

$$W_{ji} \propto O_j^1 O_i^1$$

Par mesure de simplicité, on pose généralement :

$$W_{ji} = \frac{1}{N} O_j^1 O_i^1 \tag{20.2}$$

On constate que tout vecteur $\vec{O}^k = (O_i^k)$ sera, s'il est proche de $\vec{O}^1$ dans l'espace de Hamming[3], « attiré » vers O^1. En effet, soit le terme :

$$\begin{aligned} S(\sum_i W_{ji} O_i^k) &= S(\sum_i O_j^1 O_i^1 O_i^k) \\ &= S(O_j^1) S(\sum_i O_i^1 O_i^k) \\ &= O_j^1 S(\sum_i O_i^1 O_i^k) \end{aligned}$$

Ce terme sera égal à O_j^1 si la somme $\sum_i O_i^1 O_i^k$ est positive, c'est-à-dire si le nombre de O_i^k égaux à O_i^1 est supérieure à $N/2$. Donc, pour tout vecteur initial situé à une distance de Hamming[4] inférieure à $N/2$ de O_i^1, le réseau évoluera vers $\vec{O}^1$. $\vec{O}^1$ est un *attracteur*. On voit réciproquement que, pour tout vecteur situé à une distance de Hamming supérieure à $N/2$, le réseau évoluera vers l'attracteur inverse, qui est $-\vec{O}^1$.

20.2.3 Mémorisation de plusieurs vecteurs binaires

Comment faire pour mémoriser simultanément P vecteurs binaires dans un même réseau ? Il suffit de superposer plusieurs termes de la forme 20.2, ce qui donne pour les poids :

$$W_{ji} = \frac{1}{N} \sum_{k=1}^{P} O_j^k O_i^k$$

Cette règle est appelée *règle de Hebb*, ou *règle de Hebb généralisée*. Un réseau qui utilise la règle de Hebb pour le calcul des poids des connexions et qui effectue la mise à jour des valeurs des unités de façon asynchrone est un *modèle de Hopfield*.

Il nous reste à montrer que la règle choisie pour calculer les poids vérifie bien les conditions de stabilité pour l'ensemble des vecteurs $\vec{O}^k$. Nous voulons donc :

$$\forall k,\ \forall j,\ S(\sum_i W_{ji} O_i^k) = O_j^k$$

---

3 Un espace de Hamming de dimension n est formé par l'ensemble des vecteurs $\vec{u} = (u_i)_{i=1,n}$ où u_i peut prendre deux valeurs, ici -1 ou 1.

4 La distance de Hamming de deux vecteurs $\vec{u} = (u_i)_{i=1,n}$ et $\vec{v} = (v_i)_{i=1,n}$ d'un espace de Hamming est égal au nombre de positions où les coordonnées des vecteurs $\vec{u}$ et $\vec{v}$ diffèrent.

Développons rapidement :

$$\begin{aligned} S(\sum_i W_{ji} O_i^k) &= S(\sum_i \frac{1}{N} \sum_l O_j^l O_i^l O_i^k) \\ &= S(\frac{1}{N} \sum_i O_j^k O_i^k O_i^k + \frac{1}{N} \sum_i \sum_{l \neq k} O_j^l O_i^l O_i^k) \\ &= S(O_j^k + \frac{1}{N} \sum_i \sum_{l \neq k} O_j^l O_i^l O_i^k) \end{aligned}$$

Nous voyons qu'une condition suffisante pour que ce terme soit égal à O_j^k est :

$$1 > |\frac{1}{N} \sum_i \sum_{l \neq k} O_j^l O_i^l O_i^k| \tag{20.3}$$

Ceci est en général le cas si le nombre de vecteurs stockés P est petit devant le nombre total de neurones N. Nous allons étudier le problème de la capacité de stockage dans la section suivante. Remarquons cependant que si la condition 20.3 est vérifiée, chacun des vecteurs stockés agira comme un attracteur par rapport aux vecteurs proches de lui dans l'espace de Hamming, pour des raisons absolument similaires à celles développées dans la section précédente.

20.2.4 Capacité de stockage

Nous allons dans cette section nous intéresser à la capacité de stockage d'un réseau de Hopfield, c'est à dire au nombre de vecteurs binaires différents qu'il est capable de stocker simultanément en fonction du nombre de neurones[5].

Posons :

$$T_j^k = -O_j^k \frac{1}{N} \sum_i \sum_{l \neq k} O_j^l O_i^l O_i^k$$

D'après les résultats de la partie précédente, nous savons que les vecteurs binaires sont stables si la condition suivante est vérifiée pour tout j et tout k :

$$T_j^k < 1$$

Nous allons nous intéresser au cas où tous les vecteurs $\vec{O}^k$ sont composés de bits aléatoires. Nous allons également considérer que le nombre de neurones N et le nombre de vecteurs stockés P sont grands devant 1. Nous allons tout d'abord calculer la probabilité p_e pour qu'un bit j d'un vecteur k ait une valeur incorrecte :

$$p_e = \text{Prob}(T_j^k > 1)$$

---

5 On peut trouver de plus amples résultats sur les capacités des réseaux de Hopfield dans (McEliece *et al.* 1987).

N et P étant grands devant 1, on peut approximer T_j^k à $1/N$ fois la somme de $N \times P$ nombres valant aléatoirement 1 ou -1. Il s'agit donc d'une distribution binomiale de moyenne 0 et de variance $\sigma^2 = P/N$. $N \times P$ étant grand devant 1, on peut approcher cette loi binomiale par une gaussienne. Dans ce cas, p_e se calcule facilement et vaut :

$$p_e = \frac{1}{\sqrt{2\pi}\sigma} \int_1^\infty e^{-\frac{x^2}{2\sigma^2}} dx$$

On peut calculer des valeurs numériques de p_e pour tout σ.

Réciproquement, si nous fixons p_e, il est possible de calculer la valeur de σ. Ainsi, par exemple, si l'on souhaite que la probabilité d'erreur pour un bit[6] soit inférieure à 1%, il faut prendre $\sigma = 0.185$, soit $P < 0.185N$.

Calculons maintenant la condition pour que 99% des bits soient correctement mémorisés, soit $(1 - p_e)^N > 0.99$ puisque chaque vecteur contient N bits. Comme p_e sera certainement petit devant 1, on peut faire un développement limité à l'ordre 1 et ramener cette condition à $p_e < 0.01/N$. Ceci impose que $\sigma = P/N$ tend vers 0 quand N tend vers l'infini. On peut donc faire un développement asymptotique de $\log(p_e)$ et on obtient :

$$\log(p_e) \approx -\log 2 - \frac{N}{2P} - \frac{1}{2}\log \pi - \frac{1}{2}\log(N/2P)$$

La condition $p_e < 0.01N$ devient, en passant au log :

$$\log(p_e) < \log(0.01) - \log(N)$$

Ce qui nous donne donc, en ne gardant que les termes principaux en N :

$$P < \frac{N}{2\log(N)}$$

Si l'on souhaite que tous les vecteurs binaires appris soient parfaitement restitués, il faut alors exiger que les NP bits soient restitués avec une probabilité de 99%, soit

$$p_e < \frac{0.01}{PN} \text{ donc } \frac{N}{2P} > \log(NP)$$

En approximant NP par N^2 pour N grand, on obtient :

$$P < \frac{N}{4\log(N)}$$

Si nous nous résumons, on peut donc dire que le nombre de vecteurs binaires que peut retenir un réseau dépend linéairement du nombre N de neurones (de l'ordre de $0.10 \times N$), à condition que l'on accepte un certain nombre d'erreurs dans la restitution. Si l'on souhaite une restitution proche de la perfection, alors la capacité de mémorisation est de l'ordre de $N/\log(N)$.

---

6 Il faut cependant noter que si 99% des bits sont bien stables initialement, on ne sait rien de ce qui se passera dans le réseau une fois que ces bits auront changé de valeur. Un effet d'avalanche peut parfaitement se produire, rendant l'ensemble du réseau inutilisable. On peut montrer (voir (Hertz *et al.* 1991)) que, pour éviter ce problème, on doit poser $P < 0.138N$.

20.3 Les réseaux à sens unique

Nous allons, dans cette section, nous intéresser à des réseaux constitués de couches de neurones dans lesquels la sortie d'un neurone ne peut alimenter que l'entrée d'un neurone situé dans une couche postérieure et nous allons étudier dans ces réseaux les mécanismes d'auto-apprentissage, ainsi que leurs justifications théoriques. On parle ici *d'apprentissage supervisé.*

Ces réseaux ne fonctionnent pas sur le principe de mémoire associative. Les poids des connexions sont au départ aléatoires et c'est par un mécanisme essai-erreur-correction que ce type de réseau évolue vers un état stable. Historiquement, les premiers réseaux ayant fonctionné sur ce mécanisme furent les perceptrons, développés par Rosenblatt dès 1958 (Rosenblatt 1958; 1962). Mais en 1969, Minsky et Papert (Minsky and Papert 1969) montreront que les perceptrons ne peuvent s'appliquer que pour des classes d'exemples linéairement séparables. Ainsi, un perceptron ne peut apprendre une fonction de type *OU-exclusif.* L'intérêt pour les réseaux de neurones disparut un peu, malgré les améliorations apportées au système par la règle de Widrow-Hoff. En effet, on ne savait pas réaliser l'apprentissage pour des réseaux à plus de deux couches. Cette limite disparut dans les années 80 avec la découverte d'une méthode permettant d'étendre la règle de Widrow-Hoff aux réseaux multi-couches. Pour plus de précision, on peut se reporter à (Soulié 1986) ou (Rumelhart 1986). Nous allons, dans la suite de cette section, examiner le cas des réseaux à deux couches et à couches multiples. Nous n'étudierons pas les perceptrons.

20.3.1 Les réseaux à deux couches

Nous allons examiner, comme premier exemple, les réseaux à deux couches. Dans ces réseaux, la première couche est appelée couche d'entrée et la seconde couche, couche de sortie. Chaque neurone de la couche d'entrée est raccordé à chaque neurone de la couche de sortie au moyen d'une synapse. Les neurones de sortie additionnent la somme des valeurs que leur envoient les neurones d'entrée en leur affectant un poids.

Mathématiquement, on peut noter[7] :

$$O_j = \sum_i W_{ji} I_i \tag{20.4}$$

Au départ, les poids W_{ji} sont quelconques. Le but de la méthode d'apprentissage est de les faire évoluer de façon à ce que le réseau soit capable, étant donné un vecteur d'entrée, de calculer le bon vecteur en sortie.

Le mécanisme utilisé est le suivant : on fournit au réseau un ensemble d'exemples. Chaque exemple est constitué d'un vecteur d'entrée et du vecteur de sortie correct associé. On demande alors au réseau de calculer, pour chaque vecteur d'entrée, son propre vecteur de sortie et on le compare au vecteur correct. En fonction de l'erreur, on modifie les W_{ji} jusqu'à ce que le réseau donne une réponse que l'on estime correcte (c'est-à-dire jusqu'à ce que l'erreur entre le vecteur théorique et le vecteur calculé soit inférieure à une valeur

7 Nous verrons réapparaître le seuil plus loin. Pour l'instant, nous le négligeons.

considérée comme raisonnable). Ce type de mécanisme est également appelé *apprentissage supervisé*, car on « supervise » l'apprentissage en fournissant des exemples au réseau.

Comment est calculée la modification des poids ? La formule utilisée est la suivante :

$$\Delta W_{ji} = \eta(T_j - O_j)I_i = \eta\delta_j I_i$$

η représente le coefficient d'apprentissage, T le vecteur théorique en sortie, O le vecteur calculé par le réseau en sortie et I le vecteur en entrée et $\delta_j = T_j - O_j$.

Cette règle est appelée *règle du delta* (*delta-rule* par les anglo-saxons). Nous allons maintenant montrer que cette règle est valable[8] et minimise bien l'erreur sur un exemple (I, T). Pour cela, nous allons prouver que la dérivée de la fonction d'erreur du réseau par rapport à chacun des poids est proportionnelle au changement dans les poids préconisés par la règle du delta, affectée du signe moins. Cela prouvera que la règle delta modifie les poids suivant la courbe de plus grande pente sur la surface définie par la fonction d'erreur.

Nous allons tout d'abord poser :

$$E = \frac{1}{2}\sum_j (T_j - O_j)^2 \tag{20.5}$$

E représente l'erreur entre la sortie désirée T et la sortie calculée O. Nous allons maintenant prouver que :

$$-\frac{\partial E}{\partial W_{ji}} = \delta_j I_j$$

Nous allons décomposer $\frac{\partial E}{\partial W_{ji}}$ en deux termes :

$$\frac{\partial E}{\partial W_{ji}} = \frac{\partial E}{\partial O_j}\frac{\partial O_j}{\partial W_{ji}} \tag{20.6}$$

La première partie indique comment l'erreur change quand la sortie change, et la seconde comment la sortie varie en fonction des poids. Il est maintenant facile de calculer les deux membres. Tout d'abord de l'équation 20.5 nous tirons :

$$\frac{\partial E}{\partial O_j} = -(T_j - O_j) = -\delta_j$$

Comme on pouvait s'y attendre, la contribution d'une unité à l'erreur est proportionnelle à δ_j. De plus de l'équation 20.4, nous tirons :

$$\frac{\partial O_j}{\partial W_{ji}} = I_i$$

En substituant dans l'équation 20.6,

$$\frac{\partial E}{\partial W_{ji}} = -\delta_j I_i$$

8 Il faut bien entendu que la fonction n'ait qu'un seul minimum.

QED.

Il nous reste à montrer que cette règle minimise aussi l'erreur totale pour un ensemble (I_p, T_p) d'exemples. L'erreur totale E vaut :

$$E = \sum_p E_p \tag{20.7}$$

avec , comme nous venons de le voir :

$$\frac{\partial E_p}{\partial W_{ji}} = \delta_{pj} I_{pi} \tag{20.8}$$

On a alors, à partir de 20.8 et de 20.7 :

$$\frac{\partial E}{\partial W_{ji}} = \sum_p \delta_{pj} I_{pi}$$

On voit donc que l'erreur est minimisée à condition que les poids W_{ji} ne soient pas modifiés entre chaque exemple. Or, c'est précisément ce que fait la règle du delta. C'est là qu'intervient la constante η. En effet, pour η petit, la variation des W_{ji} est négligeable par rapport à la variation de l'erreur et la méthode reste valable.

20.3.2 Les réseaux multi-couches

Un réseau multi-couches est constitué d'une couche d'entrée, d'une couche de sortie et de couches intermédiaires. Un neurone ne peut envoyer son résultat qu'à un neurone situé dans une couche postérieure à la sienne. Un vecteur d'entrée étant donné, on calcule le vecteur de sortie par passes successives à travers les couches.

Enfin, des résultats théoriques montrent que pour tirer parti de ce type de réseau, il faut utiliser des fonctions d'activation non-linéaires. Jusqu'à présent nous avions utilisé comme fonction d'activation l'identité, puisque nous écrivions[9] : $O_j = \sum_i (W_{ji} O_i)$. Désormais nous poserons :

$$N_j = \sum_i (W_{ji} O_i) \tag{20.9}$$

et

$$O_j = f_j(N_j) \tag{20.10}$$

où f est une fonction non-décroissante et différentiable.

Pour prouver que nous pouvons toujours appliquer une règle du type de la règle du delta, nous allons à nouveau nous intéresser à la dérivée de E par rapport à W_{ji} et essayer à nouveau de montrer que :

$$\Delta W_{ji} \propto -\frac{\partial E_p}{\partial W_{ji}}$$

---

9 La formule qui suit est *correcte*. Le terme O_i à droite de l'égalité était bien I_i dans le paragraphe précédent, mais dans un réseau multi-couches, l'entrée d'une couche est la sortie de la couche précédente.

en prenant pour définition de E l'équation 20.5. Nous allons appliquer la même méthode que précédemment et écrire :

$$\frac{\partial E}{\partial W_{ji}} = \frac{\partial E}{\partial N_j}\frac{\partial N_j}{\partial W_{ji}} \tag{20.11}$$

À partir de l'équation 20.9 nous voyons que le second facteur peut s'écrire :

$$\frac{\partial N_j}{\partial W_{ji}} = \frac{\partial \sum_k W_{jk}O_k}{\partial W_{ji}} = O_i$$

Nous allons maintenant définir un nouveau δ :

$$\delta_j = -\frac{\partial E}{\partial N_j}$$

Nous pouvons maintenant réécrire notre équation 20.11 :

$$-\frac{\partial E}{\partial W_{ji}} = \delta_j O_i$$

Donc, si nous voulons avoir une règle de gradient, nous devons faire en sorte que nos poids varient suivant une règle du type :

$$\Delta_j W_{ji} = \eta \delta_j O_i$$

Tout le problème est donc maintenant de voir ce que peuvent être les valeurs des δ_j pour chacune des unités u_j du réseau ; nous allons montrer qu'il est possible de calculer récursivement les δ_j en rétro-propageant les signaux d'erreur à travers le réseau.

Nous allons donc décomposer δ_j en deux termes :

$$\delta_j = -\frac{\partial E}{\partial N_j} = -\frac{\partial E_p}{\partial O_j}\frac{\partial O_j}{\partial N_j} \tag{20.12}$$

Par l'équation 20.10 on obtient :

$$\frac{\partial O_j}{\partial N_j} = f_j'(N_j)$$

Si u_j est une unité de la couche de sortie, on peut écrire comme pour un réseau bicouches :

$$\frac{\partial E}{\partial O_j} = -(T_j - O_j)$$

Nous trouvons, en remplaçant dans 20.12 :

$$\delta_j = (T_j - O_j) f_j'(N_j) \tag{20.13}$$

et ce, pour toutes les unités u_j appartenant à la couche de sortie. Pour les unités n'appartenant pas à la couche de sortie, nous calculons $\frac{\partial E}{\partial O_j}$ en le décomposant en deux termes :

$$\frac{\partial E}{\partial O_j} = \sum_k \frac{\partial E}{\partial N_k}\frac{\partial N_k}{\partial O_j} = \sum_k \frac{\partial E}{\partial N_k}\frac{\partial \sum_i W_i O_i}{\partial O_j} = \sum_k \frac{\partial E}{\partial N_k} W_{kj} = -\sum_k \delta_k W_{kj}$$

En remplaçant dans 20.12 :

$$\delta_j = f'_j(N_j)\sum_k \delta_k W_{kj}$$

Nous pouvons résumer tout cela en trois équations :

$$\Delta W_{ji} = \eta \delta_j O_i \tag{20.14}$$

$$\delta_j = (T_j - O_j) f'_j(N_j) \tag{20.15}$$

$$\delta_j = f'_j(N_j)\sum_k \delta_k W_{kj} \tag{20.16}$$

L'équation 20.14 indique la règle à appliquer pour faire varier les poids en fonction du signal d'erreur, 20.15 montre comment on calcule ce signal pour les unités des couches de sortie et 20.16 indique comment calculer ce signal par rétro-propagation pour les unités placées dans les couches cachées.

On voit donc que, dans ce cas, l'application de la règle du delta se fait en deux phases : lors de la première phase, on calcule à partir du vecteur d'entrée le vecteur de sortie. Puis on le compare au vecteur théorique et on en déduit un signal d'erreur δ_j pour chaque unité de la couche de sortie. Lors de la seconde phase, le signal d'erreur est rétro-propagé à travers le réseau et les changements dans les poids sont effectués pour chacune des couches. On a ainsi implanté une technique de rétro-propagation du gradient.

Les problèmes associés à cette méthode sont les suivants : tout d'abord, la surface définie par la fonction d'erreur dans le cas des réseaux multi-couches n'est pas concave et peut avoir un certain nombre de minima locaux ; une méthode de plus grande pente ne garantit donc pas d'atteindre le minimum absolu. Ensuite, la procédure garantit qu'il y aura apprentissage mais ne dit pas en combien de temps. Enfin, les équations montrent que cette technique préserve la symétrie des coefficients, ce qui n'est souvent pas souhaitable. Nous allons maintenant examiner quelques techniques permettant de régler (partiellement) ces problèmes.

20.4 Astuces techniques

20.4.1 La fonction d'activation

On choisit généralement comme fonction d'activation une fonction de type sigmoïde :

$$O_j = \frac{1}{1 + e^{-(\sum_i W_{ji} O_i + \theta_j)}}$$

θ_j joue ici un rôle identique à un seuil.

On remarquera que la dérivée de la fonction prend son maximum en 0.5, ce qui veut dire que ce sont les poids proches de 0.5 qui sont les plus modifiés, ou encore que les connexions dont on sait déjà qu'elles sont actives ou inactives (proches de 1.0 ou de 0.0) sont les moins modifiées. Ceci contribue à la stabilité du réseau. On remarquera également que la fonction ne peut atteindre 1.0 ou 0.0 que pour des poids infinis.

20.4.2 Le taux d'apprentissage

On constate, en regardant les équations, que plus le taux d'apprentissage est grand et plus les poids changent vite, ce qui permet un apprentissage rapide. En revanche, une méthode du gradient, pour être stable, nécessite un taux d'apprentissage aussi petit que possible. Il faut donc trouver le taux d'apprentissage le plus grand possible sans provoquer d'oscillation. Une méthode consiste à inclure un terme de momentum et de réécrire l'équation du delta :

$$\Delta W_{ji}(n+1) = \eta \delta_j O_i + \alpha \Delta W_{ji}(n)$$

α est une constante qui permet de pondérer la variation présente en fonction de la variation précédente. Cela empêche le système d'osciller avec de hautes fréquences.

On peut également faire varier le coefficient d'apprentissage et le coefficient de momentum suivant la règle suivante : on calcule un coefficient de corrélation $C(n)$ par[10] :

$$C(n) = \ll \Delta W_{ji}(n), \Delta W_{ji}(n-1) \gg$$

Si $C(n) < 0$ on prend alors :

$$\eta \leftarrow 0.01$$

$$\alpha \leftarrow 0.00$$

Si au contraire $C(n) > 0$:

$$\eta \leftarrow \eta + \delta_\eta$$

$$\alpha \leftarrow \alpha + 0.01$$

Il est parfois nécessaire de borner α et η. On peut également légèrement améliorer la règle d'apprentissage sur la variation des coefficients en n'augmentant α et η que si $C(n)$ est supérieur d'au moins 5% à sa précédente valeur. Cela stabilise l'apprentissage.

20.4.3 Comment briser la symétrie

Une méthode traditionnellement utilisée pour briser la symétrie du problème consiste à prendre au départ des poids petits mais aléatoires.

---

10 Rappel : Si A et B sont deux matrices, $\ll A, B \gg = \sum_{i,j} a_{ij} b_{ij}$

20.4.4 Comment éviter les minima locaux

Une méthode pour éviter les minima locaux est d'utiliser un recuit simulé. Le recuit simulé s'inspire, dans son principe, du phénomène de recuit en thermodynamique. Il consiste, très informellement, à introduire une imprécision autour de la courbe de la fonction énergie. De ce fait, les minima locaux, qui sont généralement des « petits creux » dans la courbe, sont « effacés » et l'on atteint ainsi le minimum global, avec une certaine imprécision. Une fois que l'on a atteint ce minimum, on réduit lentement le degré d'imprécision et par raffinement successif on atteint, avec la précision maximale, le minimum absolu[11].

20.5 Les réseaux de neurones logiques (ALN)

20.5.1 Présentation générale

L'apparition des réseaux de neurones logiques (ALN : Adaptive Logical Network en anglais) remonte au milieu des années 70. Il s'agit également de réseaux fonctionnant en apprentissage supervisé : des exemples sont présentés au réseau, une sortie est calculée, on compare la sortie calculée à la sortie théorique et on modifie en conséquence le réseau, jusqu'à ce que l'on soit satisfait de l'apprentissage. Cependant, la structure des réseaux logiques est complètement différente de celle des réseaux utilisant des techniques de rétropropagation du gradient.

Un neurone dans un ALN est une porte logique à deux entrées (gauche et droite) booléennes et une sortie booléenne. Cette porte peut être une porte *OU* dont la sortie est le résultat du OU logique entre les deux entrées, une porte *ET* (la sortie est le résultat du ET logique entre les deux entrées), une porte *DROITE* (la sortie est la valeur de l'entrée de droite) ou une porte *GAUCHE* (la sortie est la valeur de l'entrée de gauche). Un réseau de neurones logiques est un arbre binaire où chaque nœud est un neurone élémentaire. Un réseau de profondeur n calcule donc 1 bit en sortie en fonction de 2^{n-1} bits en entrée. Ainsi, le réseau représenté sur la figure 20.1 permettrait, par exemple, de calculer le bit de parité d'un octet (8 bits). L'ensemble des résultats présentés dans ce chapitre dérivent des travaux et des articles du professeur William Armstrong et de son équipe (Armstrong *et al.* 1990; Armstrong and Gecsi 1979; Armstrong *et al.* 1991).

20.5.2 Apprentissage

L'apprentissage dans les réseaux logiques se fait par la modification des fonctions des portes logiques. Au début, la valeur des fonctions des portes logiques est aléatoire. Puis, par présentation d'exemples et comparaison de la sortie théorique et de la sortie calculée, on modifie la valeur des fonctions de certaines portes. Le problème est de déterminer quelles sont les portes dont la fonction doit être modifiée, et comment la modifier.

La première notion que l'on peut définir est celle de *responsabilité*. Un nœud est dit *responsable* si le changement de sa valeur de sortie change la valeur de la sortie du réseau,

---

11 Voir les machines de Boltzmann, (Hertz *et al.* 1991), chapitre 7.1.

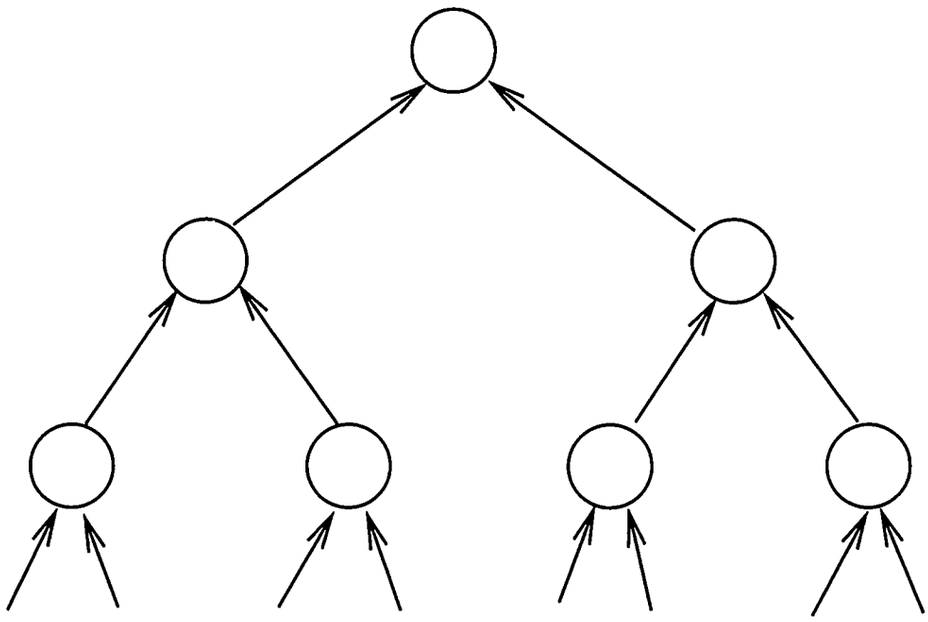


Figure 20.1 – Exemple d'ALN à 8 entrées

Fonction	$(0,0)$	$(0,1)$	$(1,0)$	$(1,1)$
ET	0	0	0	1
OU	0	1	1	1
GAUCHE	0	0	1	1
DROITE	0	1	0	1

Table 20.1 – Valeur de sortie en fonction des entrées

toutes les autres sorties restant les mêmes (sauf, bien sûr, celles dépendant directement du nœud modifié). On peut donc définir récursivement, en partant de la racine de l'arbre, la responsabilité d'un nœud : ainsi, considérons un nœud *ET* qui est responsable. Si une entrée x de ce nœud vaut 0, alors la valeur de l'entrée opposée est sans importance. En revanche, si la valeur de x est 1, alors le nœud fils relié à l'autre entrée est *responsable* de la valeur de sortie[12].

Il faut également remarquer (table 20.1) que le type de la fonction n'a de l'influence que lorsque l'entrée est $(0,1)$ ou $(1,0)$. C'est donc seulement pour ces valeurs en entrée que l'on peut *adapter* la valeur de sortie, et donc la fonction. Une technique simple consiste à associer, pour chaque neurone, à l'entrée $(0,1)$ un compteur C_{01} et à l'entrée $(1,0)$ un compteur C_{10}. En phase d'apprentissage, lorsque l'entrée est $(0,1)$, que le neurone est responsable et que la sortie théorique est 1, alors le compteur est incrémenté ; si la sortie théorique est 0, alors le compteur est décrémenté. On applique le même algorithme pour le compteur C_{10} et l'entrée $(1,0)$. À chaque étape, la sortie d'un neurone est calculée à partir de la valeur des deux compteurs. Si les compteurs sont positifs tous les deux, alors le neurone se comportera comme un *OU*. Si les deux compteurs sont négatifs, alors le neurone se comporte comme un *ET*. Si le compteur C_{01} est positif et le compteur C_{10} négatif, alors la fonction du neurone est la fonction *DROITE* ; symétriquement, si le compteur C_{10} est positif et le compteur C_{01} négatif, la fonction est *GAUCHE*.

12 Responsable ne doit pas être pris dans le sens de coupable, mais bien dans le sens de « a une influence directe sur ».

On obtient ainsi un système auto-adaptatif qui tend à récompenser les neurones qui se comportent bien et à punir les neurones qui se comportent mal[13].

Malheureusement, cette technique se révèle en pratique trop simple; elle a, en effet, tendance à entraîner le réseau vers un optimum local. D'autres techniques ont été développées, basées sur une notion de *responsabilité heuristique* d'un nœud et non plus sur la responsabilité « simple ». Elles sont décrites dans (Armstrong *et al.* 1990; Armstrong and Gecsi 1979; Armstrong 1976). On peut rapidement indiquer la plus simple : le nœud situé à la racine de l'arbre est toujours heuristiquement responsable. Si un nœud est heuristiquement responsable et qu'une de ses entrées n'est pas égale à la valeur de sortie désirée du réseau, alors cette entrée est une erreur. Si le signal de droite est une erreur, alors le nœud de gauche est heuristiquement responsable. Chaque neurone mémorise la dernière valeur du signal d'erreur pour chaque entrée ; le nœud de gauche est également responsable si le signal de droite n'est pas une erreur, mais est égal au dernier signal d'erreur pour l'entrée droite. Bien entendu, la responsabilité heuristique du nœud de droite est déterminée symétriquement. Cette stratégie est appelée par le professeur Armstrong the *latest error strategy*. Avec cette technique, on améliore considérablement les capacités d'apprentissage des réseaux.

20.5.3 Apprentissage de fonctions numériques

Les réseaux logiques ne sont pas limités à l'apprentissage de fonctions logiques. Ils peuvent également apprendre des fonctions de variables entières ou discrètes, à valeurs entières ou discrètes. La méthode naïve, qui consiste à coder en base 2 les valeurs d'entrée et de sortie, est cependant insuffisante. En effet, par un semblable mécanisme, on ne différencie pas les bits de poids faibles des bits de poids forts. Or il est clair que la présence ou l'absence d'un bit de poids fort est plus importante que la présence ou l'absence d'un bit de poids faible. Il faut donc raffiner la méthode. On commence tout d'abord par effectuer une quantification en k niveaux des valeurs d'entrée[14]. La phase suivante est un peu plus complexe; on prend la première valeur de quantification et on lui associe un vecteur aléatoire d'un espace de Hamming de dimension n. Récursivement, si j'ai associé aux $i-1$ premières valeurs de quantification, un vecteur de mon espace de Hamming, j'associe à la i-ème valeur de quantification un vecteur aléatoire du même espace de Hamming *se trouvant à une distance de Hamming* p du vecteur associé à la valeur $i-1$ (cette technique de codage est décrite dans (Smith and Stanford 1990)). L'intérêt de cette méthode est qu'elle maintient une relation de localité dans l'espace de Hamming entre deux éléments proches au sens de la distance euclidienne, tout en supprimant le problème lié aux bits de poids forts et de poids faibles. On utilisera le même mécanisme pour coder les valeurs de sortie du réseau. Il faut cependant se rappeler qu'un réseau logique ne pouvant fournir qu'un seul bit en sortie, il faudra utiliser autant de réseaux qu'il y aura de bits utilisés pour coder la variable de sortie.

---

13 On remarquera la similitude avec les techniques d'apprentissage numérique des fonctions d'évaluation pour les algorithmes de jeux. Le problème est le même : trouver dans une fonction complexe quelles sont les parties à récompenser ou à punir, en fonction du comportement global du système.

14 Par exemple, si l'on décide d'étudier la fonction x^2 sur l'intervalle $[0, 1[$, on quantifie le problème en un certain nombre de niveaux k. Si $k = 4$, les quatre valeurs 0, 1, 2 et 3 représenteront les nombres 0, 0.25, 0.50 0.75. k doit être d'autant plus grand que l'on veut une plus grande précision.

Le choix des valeurs de n et de p est complexe. n doit être suffisamment grand pour garantir l'unicité des vecteurs de l'espace de Hamming codant les niveaux de quantification, mais pas trop grand car cela augmente la taille des arbres et donc les temps de calcul. Ce problème est particulièrement critique pour le codage de la valeur de sortie, car ajouter un bit dans le codage de la variable de sortie revient à rajouter un arbre entier. De même, le choix de p doit répondre à deux contraintes. D'une part p doit être suffisamment petit pour conserver le concept de localité; deux vecteurs pris au hasard dans un espace de Hamming de dimension n ont une distance moyenne de $n/2$, donc p doit être assez nettement inférieur à $n/2$. D'autre part, il est intéressant de choisir pour la variable de sortie une valeur de p relativement grande; en effet, si la valeur de sortie calculée par le réseau est différente de tous les vecteurs servant à coder les variables de quantification, on choisit le vecteur le plus proche au sens de la distance de Hamming. Plus les vecteurs de codage sont éloignés les uns des et autres et plus le décodage basé sur la méthode de distance minimale est efficace.

20.5.4 Avantages des ALN

Les ALN présentent un certain nombre d'avantages par rapport à l'approche des réseaux à rétro-propagation du gradient :

Apprentissage paresseux : on n'a besoin de s'intéresser qu'aux parties de l'arbre qui sont responsables. Les autres sont sans intérêt, d'où un gain de temps certain.

Évaluation paresseuse : lorsque le réseau est appris, on peut utiliser des techniques d'évaluation paresseuse pour améliorer la vitesse de calcul. Ainsi, lorsque l'on a calculé la valeur de l'entrée gauche d'un neurone *ET*, et que cette valeur est 0, il est inutile d'évaluer le sous-arbre de droite. De même pour une porte *OU*, si la valeur est 1, alors il est également inutile d'évaluer le sous-arbre de droite. On gagne ainsi un temps précieux dans l'évaluation.

Pouvoir de représentation : les ALN ont la capacité de pouvoir représenter sans difficulté des fonctions présentant des discontinuités, ou dont la dérivée présente une discontinuité. Les réseaux utilisant des sigmoïdes « arrondissent » les discontinuités des fonctions.

Possibilité d'implantation hardware : une fois qu'un apprentissage est terminé, il est possible de construire un réseau électronique reprenant l'architecture logique apprise. De semblables réseaux ont des vitesses de calculs incomparablement supérieures à n'importe quel autre système, le calcul d'une valeur s'évaluant en nanosecondes !

Insensibilité aux perturbations : une des caractéristiques essentielles des ALN est leur insensibilité aux perturbations. Supposons que nous disposions d'un réseau aléatoire et que nous introduisions un bruit sur chacune des entrées *i.e.*, la probabilité qu'un bit soit changé en son opposé est p. Quelle est l'influence sur la couche suivante ? Il est aisé de prouver que la probabilité est $p - p^2/4$. En effet, la probabilité de changement pour une porte *ET* ou *OU* est $p - p^2/2$ et pour une porte *DROITE* ou *GAUCHE* est p[15]. Donc, la probabilité que la valeur d'un bit soit changé dans

15 Il est facile de vérifier le résultat en énumérant les conditions de changement de la valeur d'un bit en sortie, en fonction de la valeur des bits en entrée.

	2	4	6	8	10	12	14	16	18	20
1.0	0.60	0.45	0.36	0.30	0.26	0.23	0.20	0.18	0.17	0.15
0.8	0.54	0.41	0.33	0.28	0.24	0.22	0.19	0.18	0.16	0.15
0.6	0.44	0.36	0.30	0.26	0.23	0.20	0.18	0.17	0.15	0.14
0.4	0.33	0.28	0.24	0.21	0.19	0.17	0.16	0.15	0.14	0.13
0.2	0.18	0.16	0.15	0.14	0.13	0.12	0.12	0.11	0.10	0.10

Table 20.2 – Mesure d'insensibilité d'un ALN

la couche n est donnée par la loi : $p_{n+1} = p_n - p_n^2/4$ avec $p_0 = p$. On peut voir dans la table 20.2 une évaluation numérique de cette loi (en abscisse, nombre de couches, en ordonnée probabilité p de modification de chaque bit dans la couche d'entrée). On constate donc que les ALN sont très insensibles au bruit, et que cette insensibilité est d'autant plus grande que l'arbre est profond. Or, la reconnaissance de forme s'appuie sur cette propriété fondamentale : une petite perturbation des entrées doit peu modifier la sortie. L'insensibilité remplace dans les ALN la continuité que l'on trouve dans les réseaux à rétro-propagation.

L'approche des ALN est relativement peu connue aujourd'hui et mérite pourtant une grande attention. Ils permettent de réaliser des implantations rapides et des simulations efficaces, car uniquement en nombres entiers. Nous nous sommes attardés sur ce sujet car il est peu décrit dans la littérature classique, particulièrement en langue française. Espérons que cet oubli sera rapidement réparé.

20.6 Les réseaux de Kohonen

20.6.1 Principes généraux

Les réseaux de Kohonen sont des réseaux à *apprentissage non supervisé*. Nous n'en dirons qu'un mot rapide. On peut se reporter pour plus de détails à (Kohonen 1988b).

Dans un réseau de Kohonen, il n'y a pas apprentissage par présentation d'exemple, mais auto-adaptation à des stimuli d'entrée. Le réseau est constitué de deux couches seulement. Chaque neurone de la couche d'entrée alimente tous les neurones de la couche de sortie, et tous les neurones de la couche de sortie sont connectés entre eux (figure 20.2). Un *seul* neurone de la couche de sortie est actif lorsqu'une entrée est présentée. Le principe général d'un réseau de Kohonen est de faire une classification des entrées et de fournir une sortie qui soit la même pour des entrées proches.

20.6.2 Calcul de la valeur de sortie

Notons x_i les valeurs en provenance de la couche d'entrée (couche 1). On pose pour chaque neurone j de la couche de sortie (couche 2) :

$$E_j(n) = \sum_{i \in \text{couche 1}} W_{ij} x_i \qquad (20.17)$$

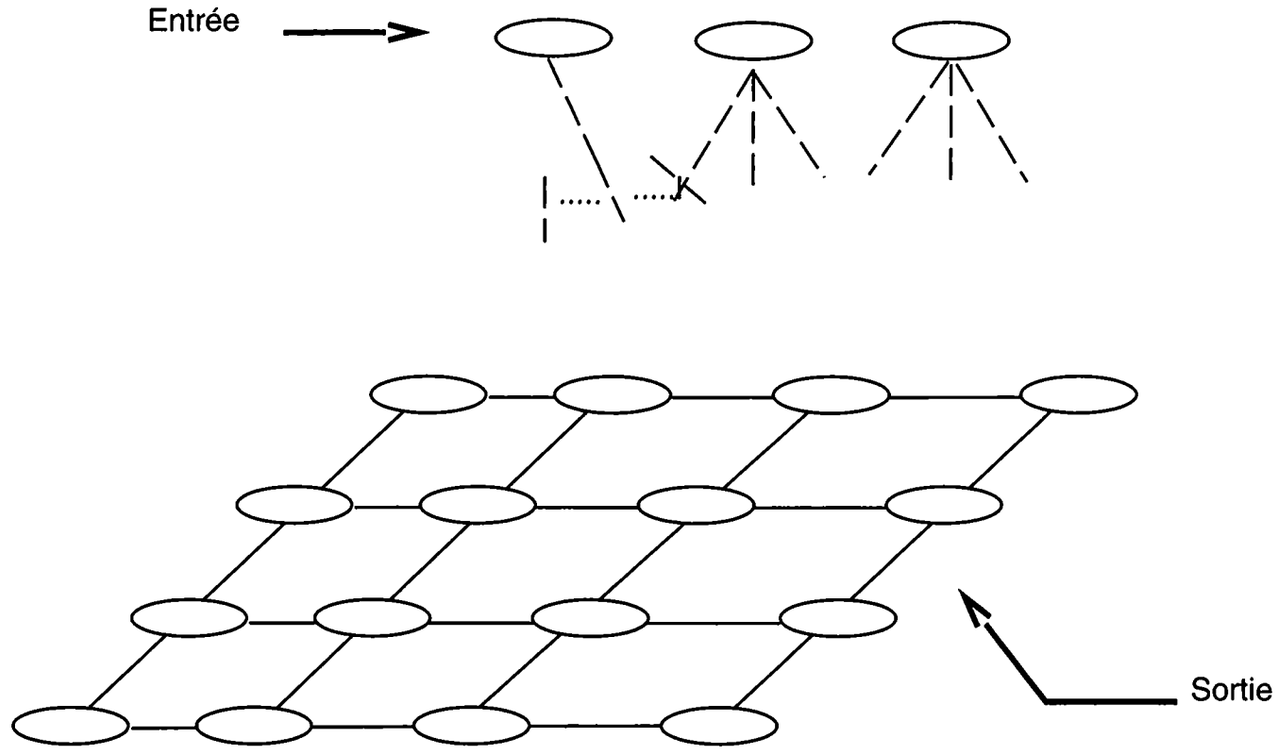


Figure 20.2 – Carte de Kohonen

$$V_j(n) = \sum_{k \in \text{couche 2}} A(j,k) S_k(n-1) \tag{20.18}$$

$$S_j(n) = f(E_j(n) + V_j(n)) \tag{20.19}$$

$$D_j = \sqrt{\sum_{i \in \text{couche 1}} (x_i - W_{ij})^2} \tag{20.20}$$

$$p \quad \text{tel que} \quad D_p = \min_{k \in \text{couche 2}} D_k \tag{20.21}$$

W_{ij} : matrice des poids des connexions entre la couche 1 et la couche 2.

$A(j,k)$: fonction d'atténuation. La valeur de $A(j,k)$ est fonction de la distance $d(j,k)$ du neurone j au neurone k dans la carte de sortie. La valeur de $A(j,k)$ en fonction de la distance d est représentée sur la figure 20.3. La fonction renforce l'influence des neurones très proches, considère les neurones plus éloignés comme inhibiteurs, les neurones très éloignés n'ayant aucune influence.

$f(x)$: fonction sigmoïde.

$E_j(n)$: influence des neurones de la couche d'entrée sur le neurone j de la couche de sortie à l'étape n.

$V_j(n)$: influence des autres neurones de la couche de sortie sur le neurone j de la même couche à l'étape n.

$S_j(n)$: état du neurone j de la couche de sortie à l'étape n.

D_j : distance quadratique entre le vecteur d'entrée et le vecteur pondération.

p : neurone qui sera actif. p est le neurone pour lequel la distance, entre le vecteur pondération et le vecteur d'entrée, est la plus faible.

Le mécanisme est donc très simple. On présente au réseau un exemple et il calcule quel neurone en sortie sera actif.

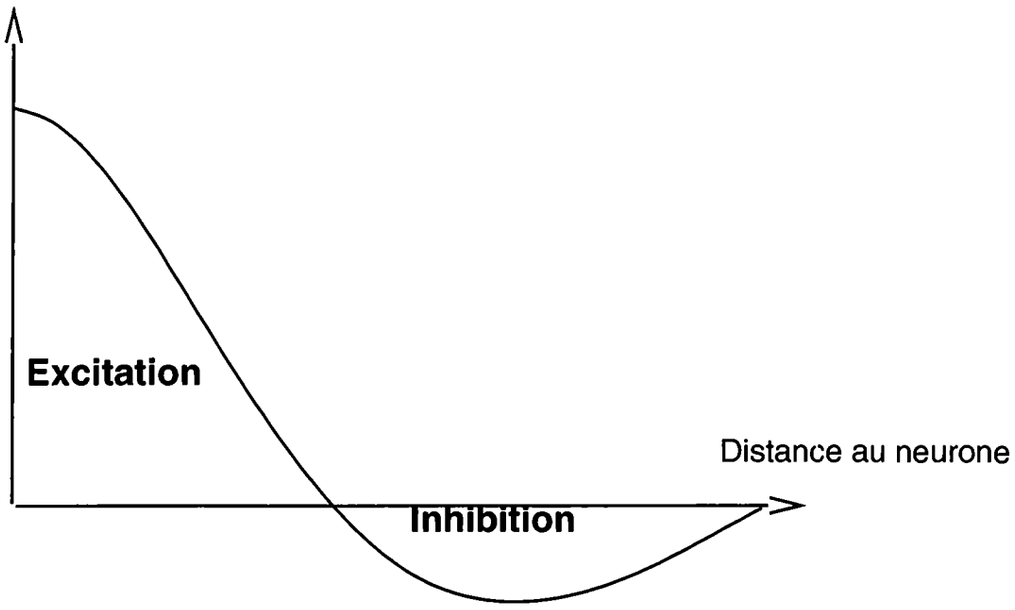


Figure 20.3 – Fonction d'atténuation

20.6.3 Apprentissage

L'apprentissage a pour but d'améliorer les capacités de discrimination du réseau. La formule de modification des poids des coefficients est :

$$W_{ij}(k) = W_{ij}(k-1) + \eta(x_i - W_{ij}(k-1))$$

Seul le neurone excité et ses voisins modifient leur poids. Les voisins d'un neurone j sont les neurones se trouvant à l'intérieur d'un cercle de rayon R autour de j.

Au début de l'apprentissage, les valeurs de R et η sont relativement grandes. On diminue ces valeurs pendant l'apprentissage de façon à rendre le réseau aussi discriminant que possible.

20.7 Conclusion

Depuis quelques années, les réseaux de neurones ont connu une popularité croissante, pour devenir aujourd'hui un des sujets de recherche les plus courus. Il faut noter que l'approche connexionniste est complètement différente de l'approche traditionnelle. L'approche classique consiste à trouver un algorithme performant et à utiliser le langage et le processeur adaptés. Elle conduit à affirmer : *je dispose d'une solution algorithmique du problème P qui nécessite n opérations*. Le mécanisme connexionniste conduit, au contraire, à des propositions du style : *l'architecture A du réseau d'automates de type T, agrémentée de la formule F de calcul des coefficients d'interaction permet de résoudre le problème P.*

L'approche connexionniste semble bien adaptée à des problèmes d'optimisation ou de reconnaissance de formes. Des exemples d'utilisations réussies de réseaux de neurones ont été donnés par Hopfield et Tank en optimisation (programmation linéaire par exemple), électronique (conversion analogique numérique), traitement du signal, et par d'autres

équipes dans le domaine des jeux (Tic-tac-toe, Backgammon), de la reconnaissance de caractères, et des systèmes experts (diagnostic médical).

L'approche connexionniste présente plusieurs avantages : le premier est que la connaissance est distribuée dans le réseau et que la panne d'une partie du réseau n'empêche pas forcément l'ensemble du système de fonctionner (quand il ne s'agit pas d'une simulation sur une machine classique, ce qui est en général le cas).

Surtout, ces systèmes ont une capacité à la généralisation[16]. Comme l'ont montré plusieurs équipes, lorsque le réseau a appris certaines formes (des lettres par exemple), il est capable de reconnaître ces mêmes formes légèrement déformées, sans qu'il soit besoin de refaire un apprentissage. Enfin, la solution d'un problème peut, pour certains types de réseaux, être donnée de façon quasi-instantanée, puisqu'il suffit d'effectuer une passe à travers le réseau.

Les réseaux neuronaux ne résoudront pas, du jour au lendemain, l'ensemble des problèmes de l'informatique. En particulier, on doit toujours prendre soin de comparer les résultats obtenus avec les réseaux de neurones formels et les résultats obtenus par des méthodes classiques (statistiques par exemple) et se méfier des phénomènes de mode qui accompagnent chaque nouvelle méthode. Il s'agit cependant d'un champ de recherche complètement nouveau, et d'une approche absolument différente qui mérite d'être examinée avec intérêt.

Leur développement ne pourra se faire que par un développement conjoint des méthodes théoriques et des moyens de calcul (circuits VLSI adaptés). Le nombre croissant de conférences et de publications, aussi bien dans les revues d'informatique que d'électronique, semble montrer que le sujet est pris très au sérieux actuellement.

16 Cependant, la faculté de généralisation est essentiellement empirique : rien ne garantit que la généralisation obtenue correspond bien à la généralisation souhaitée.

CHAPITRE 21

Algorithmes évolutionnaires

Nicolas Durand – Jean-Marc Alliot

21.1 Introduction

Les algorithmes évolutionnaires sont des algorithmes d'optimisation s'appuyant sur des techniques dérivées de la génétique et de l'évolution naturelle[1] : croisements, mutations...

Encore peu connus en France[2], leur popularité est croissante aux États-Unis surtout depuis 1985.

Ce sont des méthodes d'optimisation stochastique qui ne requièrent *a priori* pas de régularité sur les fonctions optimisées : les applications possibles sont très diverses. Ils ont la propriété de pouvoir localiser plusieurs optima mais présentent deux inconvénients majeurs. D'une part, ce sont des méthodes chères en calculs qui ne garantissent pas l'optimalité du résultat et l'on conseille généralement d'appliquer ce type de méthodes lorsque *rien d'autre ne marche*. D'autre part, la mise en oeuvre pratique de ces méthodes demande une grande expérience et les résultats théoriques sont généralement peu utiles en pratique.

On distingue quatre familles historiques :

- les algorithmes génétiques sont apparus aux Etats-Unis dans les années 60 : les premiers travaux de John Holland sur les systèmes adaptatifs remontent à 1962 (Holland 1962) ;
- la programmation génétique a d'abord été considérée comme un sous-domaine des algorithmes génétiques, c'est en train de devenir un domaine de recherche à part entière. Son défi est *d'apprendre à la machine à exécuter des tâches sans qu'elle ait été explicitement programmée pour cela.*
- Les stratégies d'évolution sont nées en Allemagne dans les années 60 (Fogel 1962) pour résoudre des problèmes d'optimisation numérique.
- La programmation évolutionnaire, apparue dans les années 60 en Californie, d'abord appliquée sur des machines à états finis.

---

1 Il est intéressant de trouver dans l'œuvre d'un spécialiste de zoologie, Richard Dawkins (Dawkins 1989), un exemple informatique tendant à prouver la correction de l'hypothèse darwinienne de la sélection naturelle. La méthode utilisée est presque semblable aux techniques d'algorithmes génétiques.

2 Dans (Jones 1992), on trouve une présentation des techniques auto-évolutives, un sujet assez proche des algorithmes génétiques.

21.2 Algorithmes génétiques

21.2.1 Principe général

Un algorithme génétique réalise une optimisation dans un espace de données. Pour pouvoir utiliser les techniques génétiques il faut disposer de deux opérations :

1. une fonction de codage de la ou des données en entrée : originellement les données étaient codées sous forme d'une séquence de bits, d'autres forme de codages sont utilisées en pratique (voir paragraphe 21.5.1) ;
2. fournir une fonction d'utilité, d'adaptation ou *fitness* $U(x)$, permettant de calculer l'adaptation d'une séquence de bits x. Idéalement, cette fonction vaudra 1 si la séquence de bits est parfaitement adaptée et 0 si la séquence est inadaptée. Cette fonction d'utilité correspond au critère à maximiser sur l'espace de données.

Ces deux éléments sont les seuls éléments spécifiques au problème à résoudre. Une fois qu'ils sont fixés, l'algorithme génétique que l'on appliquera sera toujours le même, que l'on optimise la distribution de gaz dans un pipeline, la trajectoire de deux mobiles qui doivent s'éviter, ou la fonction énergie d'un réseau de neurones.

Il s'exécutera en plusieurs étapes :

1. génération aléatoire d'un certain nombre de séquences de bits pour composer la « population » initiale ;
2. mesure de l'adaptation de chacune des séquences présentes ;
3. reproduction de chaque séquence en fonction de son adaptation. Les séquences les mieux adaptées se reproduisent mieux que les séquences inadaptées. La nouvelle population est composée des séquences après reproduction ;
4. on remplace un certain nombre de paires de séquences tirées aléatoirement par le croisement de ces paires (le lieu de croisement dans la séquence de bits est également choisi de façon aléatoire.). Chaque nouvelle paire est constituée de la façon suivante :
   - une séquence est composée de la première partie de la première séquence et de la seconde partie de la seconde séquence ;
   - l'autre séquence est composée de la première partie de la seconde séquence et de la seconde partie de la première séquence.
5. mutation d'un bit choisi aléatoirement dans une ou plusieurs séquences tirées au sort ;
6. Retour à l'étape 2.

Nous allons voir, dans le détail et sur un exemple, comment fonctionne un algorithme génétique.

21.2.2 Un exemple

Nous allons considérer l'exemple suivant : soit la fonction[3] $f(x) = 4x(1-x)$ prenant ses valeurs sur $[0, 1[$. Nous souhaitons trouver le maximum de cette fonction.

---

3 Cet exemple n'est certainement pas le meilleur que l'on puisse prendre, pour bien des raisons. Goldberg présente quant à lui la maximisation de x^2, mais il nous a semblé plus intéressant de trouver le minimum d'une fonction dont l'extremum n'est pas une borne de l'intervalle.

Séquence	Valeur	$U(x)$	% de chance de reproduction	% cumulés	Après reproduction
10111010	0.7265625	0.794678	0.794678 / 2.595947 = 0.31	0.31	11011110
11011110	0.8671875	0.460693	0.460693 / 2.595947 = 0.18	0.31+0.18=0.49	10111010
00011010	0.1015625	0.364990	0.364990 / 2.595947 = 0.14	0.49+0.14=0.63	01101100
01101100	0.4218750	0.975586	0.975586 / 2.595947 = 0.37	0.62+0.37=1.00	01101100
=		2.595947			

Table 21.1 – Population initiale et adaptation au milieu

Pour coder les données, on peut se contenter de multiplier x par 2^8 et de prendre la partie entière du résultat. On obtient ainsi un nombre entier appartenant à l'intervalle $[0, 2^8 - 1]$. Sa représentation en binaire donne une séquence de 8 bits[4] le codant. Ainsi $x = 0.3$ multiplié par 2^8 donne 76.8, dont on prend la partie entière : 76. 76 en binaire donne : 01001100. C'est la séquence de bits codant 0.3.

La fonction $U(x)$ est ici : $U(x) = f(x)$. Il s'agit donc exactement de la fonction à maximiser.

Les deux fonctions étant fixées, il ne reste qu'à appliquer l'algorithme génétique.

Première étape : nous commençons par générer aléatoirement les séquences qui vont composer notre population. Nous ne générons que 4 séquences pour cet exemple particulier[5]. Nous présentons dans le tableau 21.1 la population initiale générée, ainsi que la valeur de la fonction d'adaptation et le pourcentage de reproduction pour chaque séquence dans les trois premières colonnes.

Nous devons ensuite effectuer la deuxième étape, la reproduction. Pour savoir quels sont les quatre éléments de la population suivante avant croisement, on tire aléatoirement quatre nombres entre 0 et 1. La comparaison du nombre avec le pourcentage cumulé de la quatrième colonne du tableau 21.1 permet de savoir quel est l'élément généré. Si nous tirons les 4 nombres 0.47, 0.18, 0.89, 0.75, nous obtenons la colonne 6 du tableau comme population après reproduction.

Passons maintenant au croisement. Un certain nombre de séquences de la population vont être remplacées par des croisements avec d'autres membres de cette population. Typiquement, de 25 à 75% des séquences sont laissées intactes, les autres étant remplacées par des croisements. Supposons que le sort désigne les séquences 1 et 3 comme devant être croisées et remplacées par les résultats du croisement. Il faut encore tirer aléatoirement un nombre entre 1 et 7 qui va désigner l'endroit où le croisement va s'effectuer ; supposons que ce soit 5. Le croisement est alors représenté sur la figure 21.1.

La population après croisement est représentée dans le tableau 21.2. Il resterait à appliquer les mutations. Mais le pourcentage de mutation est toujours très faible (inférieur ou très inférieur à 10%) et nous supposons qu'il n'y en a pas ici.

Nous constatons que non seulement la moyenne des valeurs d'adaptation est meilleure, mais que la qualité de la meilleure séquence est plus grande.

---

4 Nous aurions pu multiplier par 2^n et nous aurions alors obtenu une séquence de n bits. La valeur de n que l'on choisit dépend, bien entendu, de la précision que l'on souhaite obtenir.

5 Ceci est très peu. Dans un cas réel, la population est beaucoup plus importante ; plusieurs centaines de séquences est un nombre raisonnable.

$$\begin{array}{r} 01101|100 \\ 11011|110 \\ \hline 01101|110 \\ 11011|100 \end{array}$$

Figure 21.1 – Croisement des chaînes 3 et 1

Après croisement	Valeur	$U(x)$
01101110	0.4296875	0.980225
10111010	0.7265625	0.794678
11011100	0.8593750	0.483398
01101100	0.4218750	0.975586

Table 21.2 – Population finale

21.2.3 Discussions des divers paramètres

Comme souvent pour ce type de mécanisme qui pourrait apparaître comme un peu « magique » (les réseaux de neurones font partie de cette catégorie), les difficultés sont invisibles, mais bien réelles. La première d'entre elles consiste à bien choisir les divers paramètres : pourcentage de croisement, pourcentage de mutation, taille de la population...

Mais le problème le plus important est le codage des données. Comme le dit Goldberg lui-même, le codage des données est un art, et de cet art dépend le succès ou l'échec de la tentative. Dans un cas aussi simple que celui-ci, le codage parait évident et pourtant, on remarquera que les séquences 10000000 et 01111111 sont très bien adaptées l'une et l'autre, bien que leur codage soit fondamentalement différent, ce qui n'est pas une bonne propriété[6]. La théorie des graphes présentée dans la section 21.3.2 permet de comprendre ce qu'est un bon codage de données.

21.3 Convergence théorique

21.3.1 Généralités

Plusieurs approches ont été envisagées. La théorie des schémas présentée dans la section 21.3.2 est la plus simple à aborder et donne dans un cadre simplifié une idée du mécanisme de convergence des AGs. On trouve néanmoins plusieurs démonstrations de convergence théorique des algorithmes évolutionnaires dont :

- Un résultat très général de convergence globale proposé par Zhigljavsky (Zhigljavsky 1991), au sens de la convergence faible des mesures de probabilités.

6 Une technique classique, déjà rencontrée pour les ALN, consiste à utiliser des vecteurs d'un espace de Hamming (Schraudolf and Grefenstette 1991).

- Des résultats partiels basés sur un modèle des algorithmes évolutionnaires en tant que chaînes de Markov dans l'espace des populations (Davis and Principe 1991; 1993; Nix and Vose 1992; Fogel *et al.* 1966).
- Un résultat de convergence fort pour les AGs basé sur la théorie de Friedlin Weinzel des perturbations stochastiques des systèmes dynamiques (Cerf 1994). Cette dernière approche est la plus satisfaisante tant sur le plan mathématique, que sur celui de la modélisation, les différents opérateurs étant présentés comme "perturbant" un processus Markovien représentant la population à chaque étape. Ici encore il est démontré l'importance de l'opérateur de mutation, le croisement pouvant être totalement absent. L'investissement nécessaire pour la compréhension des démonstrations de Cerf est assez important, le lecteur intéressé se reportera aux références citées. De plus, de nombreuses interrogations demeurent cependant concernant les relations réelles entre les différents paramètres caractérisant l'algorithme génétique et les choix pratiques de ceux-ci. Dans ce domaine, la pratique devance encore la théorie, même si les mécanismes commencent à être plus clairs. Les raffinements utilisés en pratique pour faire converger l'AG (sharing, scaling, clustering) n'ont pas encore été étudiés sur le plan théorique.
- Des résultats de convergence globaux avec taux de convergence pour certains algorithmes de stratégies d'évolution, et sur des fonctions convexes (Back *et al.* 1993). Le fait que ces résultats soit obtenus sur des fonctions convexes appauvrit largement leur intérêt, sachant que pour de telles fonctions, des algorithmes classiques de type gradient sont généralement largement plus performants.

21.3.2 Théorie des schémas

Définitions fondamentales

Nous allons d'abord poser quelques définitions :

Définition 21.1 – Séquence – *On appelle séquence A de longueur $l(A)$ une suite $A = a_1 a_2 \ldots a_l$ avec $\forall i \in [1, l], a_i \in V = \{0, 1\}$.*

Ceci correspond à la notion de séquence de bits que nous avons utilisée jusqu'à présent.

Nous avons également besoin de la notion de schéma et de quelques définitions associées :

Définition 21.2 – Schéma – *On appelle schéma H de longueur l une suite $H = a_1 a_2 \ldots a_l$ avec $\forall i \in [1, l], a_i \in V^+ = \{0, 1, *\}$.*

Comme nous allons le voir, une $*$ en position i signifie que a_i peut être indifféremment un 0 ou un 1.

Définition 21.3 – Instance – *On dit qu'une séquence $A = a_1 \ldots a_l$ est une instance d'un schéma $H = b_1 \ldots b_l$ si pour tout i tel que $b_i \neq *$ on a $a_i = b_i$.*

Ainsi, $H = 010 * 0101$ est un schéma et les séquences 01000101 et 01010101 sont des instances de H (ce sont même les seules).

Définition 21.4 – Position fixe, position libre – *Soit un schéma H. On dit que i est une position fixe de H si $a_i = 1$ ou $a_i = 0$. On dira que i est une position libre de H si $a_i = *$.*

Définition 21.5 – Ordre d'un schéma – *On appelle ordre du schéma H le nombre de positions fixes de H. On note l'ordre de H $o(H)$.*

Par exemple, le schéma $H = 01**10*1$ a pour ordre $o(H) = 5$, le schéma $H' = *****101$ a pour ordre $o(H') = 3$. On remarquera qu'un schéma H de longueur $l(H)$ et d'ordre $o(H)$ admet $2^{(l(H)-o(H))}$ instances différentes.

Définition 21.6 – Longueur fondamentale – *On appelle longueur fondamentale du schéma H la distance séparant la première position fixe de H de la dernière position fixe de H. On note cette longueur fondamentale $\delta(H)$.*

Ainsi, le schéma $H = 1**01***$ a pour longueur fondamentale $\delta(H) = 5-1 = 4$, le schéma $H' = 1******1$ a pour longueur fondamentale $\delta(H') = 8 - 1 = 7$, et pour le schéma $H\ >>= **1*****$ nous avons $\delta(H\ >>) = 3 - 3 = 0$.

Définition 21.7 – Adaptation d'une séquence – *On appelle adaptation d'une séquence A une valeur positive que nous noterons $f(A)$.*

f est ce que nous avions appelé jusqu'à présent la fonction d'utilité[7].

Définition 21.8 – Adaptation d'un schéma – *On appelle adaptation d'un schéma H la valeur*

$$f(H) = \frac{\sum_{i=1}^{2^{(l(H)-o(H))}} f(A_i)}{2^{(l(H)-o(H))}}$$

où les A_i décrivent l'ensemble des instances de H.

L'adaptation d'un schéma est donc la moyenne des adaptations de ses instances.

Effets de la reproduction

Soit un ensemble $S = \{A_1, \ldots, A_i, \ldots, A_n\}$ de n séquences de bits tirées aléatoirement. Durant la reproduction, chaque séquence A_i se reproduira avec une probabilité :

$$p_i = \frac{f(A_i)}{\sum_{i=1}^{n} f(A_i)}$$

Supposons que nous ayons à l'instant t un nombre $m(H,t)$ de séquences représentant le schéma H dans la population S. À l'instant $t+1$ nous devons statistiquement avoir un nombre :

$$m(H,t+1) = m(H,t).n.\frac{f(H)}{\sum_{i=1}^{n} f(A_i)}$$

Posons

$$\overline{f_t} = \frac{\sum_{i=1}^{n} f(A_i)}{n}$$

$\overline{f_t}$ représente la moyenne de l'adaptation des séquences à l'instant t. La formule précédente devient :

$$m(H,t+1) = m(H,t)\frac{f(H)}{\overline{f_t}}$$

---

7 Le « f » de f ne vient pas de fonction, mais de l'anglais « fitness ».

Posons $c_t(H) = \frac{f(H)}{\overline{f_t}} - 1$. On obtient alors :

$$m(H, t+1) = (1 + c_t(H))m(H, t)$$

On voit donc qu'un schéma, dont l'adaptation est au-dessus de la moyenne, voit son nombre de représentants augmenter, suivant une progression qui est de type géométrique si nous faisons l'approximation que $c_t(H)$ est constant dans le temps. Nous avons alors :

$$m(H, t) = (1 + c(H))^t.m(H, 0)$$

Le résultat est donc clair : si la reproduction seule était en jeu, les schémas forts éliminaient très rapidement les schémas faibles.

Effets des croisements

Nous allons nous intéresser à la probabilité de survie $p_s(H)$ d'un schéma H lors d'une opération de croisement. Considérons cela sur un exemple. Soit le schéma $H = **10*1**$. Supposons qu'une séquence qui est une instance de ce schéma soit croisée avec une autre séquence. Quelle est la probabilité pour que la séquence résultante soit encore une instance de H ? Il est impossible de répondre exactement à la question, tout au plus peut-on donner une borne inférieure de cette valeur. Il est clair que H ne sera pas détruit si le site de croisement qui est tiré au sort est inférieur à 3 (avant le premier 1) ou s'il est supérieur à 5 (après le dernier 1)[8].

On voit donc immédiatement qu'une borne inférieure de la probabilité de détruire un schéma H est $\delta(H)/(l-1)$. Donc la probabilité de survie dans un croisement est $1 - \delta(H)/(l-1)$. Si d'autre part on ne croise qu'une fraction p_c de séquences dans une population donnée, la probabilité de survie est donnée par :

$$p_s \geq 1 - p_c \frac{\delta(H)}{l-1}$$

En rassemblant ce résultat et celui du paragraphe précédent, nous obtenons pour l'évolution d'une population[9] :

$$m(H, t+1) \geq m(H, t)(1 + c_t(H))(1 - p_c \frac{\delta(H)}{l-1})$$

Effets des mutations

Supposons que la probabilité de mutation d'un bit dans une séquence soit p_m. Dans un schéma H, seules les positions fixes peuvent être détruites. Comme la probabilité de survie d'un bit est $1 - p_m$, la probabilité de survie d'un schéma H contenant $o(H)$ positions fixes est $(1 - p_m)^{o(H)}$. La probabilité de mutation étant toujours petite devant 1, on peut faire un développement limité au premier ordre, ce qui nous donne une probabilité de survie de $1 - o(H)p_m$.

En introduisant ce terme, nous avons donc l'équation finale :

$$m(H, t+1) \geq m(H, t)(1 + c_t(H))(1 - p_c \frac{\delta(H)}{l-1} - o(H)p_m)$$

---

8 Dans tous les autres cas, H peut être (ou ne pas être) détruit.

9 On rappelle que les croisements et les mutations se font sur la population reproduite, et non sur la population initiale.

Conclusions

La formule ci-dessus nous apprend deux choses :
- les schémas qui ont une longueur fondamentale petite sont plus favorisés que les autres, lors de la génération d'une nouvelle population ;
- les schémas qui ont un ordre petit sont plus favorisés que les autres, lors de la génération d'une nouvelle population.

Ceci enseigne une chose fondamentale pour le codage de données : les schémas qui codent les données « intéressantes » pour le problème doivent avoir un ordre et une longueur fondamentale faibles[10], alors que les données « sans intérêt » doivent être codées par des schémas qui ont un ordre et une longueur fondamentale élevés.

Les résultats présentés ci-dessus ne sont qu'une introduction à la théorie des schémas, il existe de nombreux approfondissements. On trouvera dans les références suivantes les principaux modèles théoriques sur la théorie des schémas et ses extensions :(Goldberg 1989b; Vose 1991; Bui and Moon 1993; Goldberg 1989a; Bridges and Goldberg 1987; Caruana and Schaffer 1988; Moon 1994; Dasgupta and McGregor 1992; Smith *et al.* 1991)

21.4 Exemple complet d'application

21.4.1 Principe

Nous allons tester l'utilisation des algorithmes génétiques sur un cas très simplifié d'optimisation de trajectoires d'avions dans le plan horizontal. Pour plus détails, on peut se reporter à (Durand 1996). Nous posons comme hypothèses :
- à chaque avion est assigné un point de départ et d'arrivée situé à 20 minutes de vol du point de départ ;
- à tout instant les avions doivent se trouver à une distance l'un de l'autre d'au moins 15km (on suppose que c'est le cas à $t = 0$) ;
- les n avions naviguent à vitesse constante ;
- chaque avion peut effectuer au plus une altération de cap de 10, 20 ou 30 degrés à droite ou à gauche à un instant t_0. Il prend ensuite un cap direct vers sa destination à un instant t_1 ;
- on cherche à minimiser le nombre de manœuvres données aux avions, et à nombre de manœuvre constant, minimiser le retard moyen des avions à l'arrivée engendré par les manœuvres.

21.4.2 Modélisation du problème

Un avion est ici défini comme un point matériel muni d'un vecteur vitesse (dans l'exemple qui suit, les avions volent à 500 nœuds ou 900 km/heure). Nous nous sommes limités au plan horizontal car, en contrôle aérien, on préfère donner aux avions des manœuvres dans le plan horizontal plutôt que des manœuvres de montée ou de descente

10 Les données intéressantes sont bien entendu les données qui sont proches de la solution. Un bon codage des données implique donc d'avoir une *idée* de la forme de la solution.

(leur coût en termes de consommation et d'exécution est plus faible, le confort du passager est préservé). Le temps est discrétisé par pas de 10 secondes.

21.4.3 Application de l'algorithme génétique

Le codage binaire présenté dans la section 21.2.2 pourrait être utilisé dans le cas présent, mais on lui préférera un codage plus immédiat qui préserve la structure des données. La manœuvre effectuée par un avion est déterminée par un triplet (a, t_0, t_1) où $a \in \{-30, -20, -10, 0, 10, 20, 30\}$, t_0 et t_1 sont des variables réelles comprises entre 0 et 20 minutes. C'est ce triplet qui sera retenu comme codage, les opérateurs de croisement et de mutation devront être adaptés en conséquence. Un problème à n avions sera codé par un $3n$ -uplet.

Le critère d'évaluation retenu s'exprime de la façon empirique suivante :

$$\begin{aligned} f &= \frac{1}{2+C} \quad si\ C > 0 \\ &= \frac{1}{2} + \frac{1}{1+N+R} \quad si\ C = 0 \end{aligned}$$

où C est la durée totale de conflits (somme des durées des périodes pendant lesquelles deux avions sont à moins de 15 km l'un de l'autre), N est le nombre de manœuvres (on considère qu'il y a manœuvre si $a \neq 0$ et $t_0 < t_1$) et R est le retard moyen à l'arrivée relatif à la durée du vol (20 minutes). D'autres choix sont possibles, celui-ci garantie qu'une solution sans conflit ($> \frac{1}{2}$) est toujours plus adaptée qu'une solution avec conflit ($< \frac{1}{2}$). Par ailleurs, N est entier et $R < 1$ donc le critère d'évaluation minimise le nombre de manœuvres avant de minimiser le retard.

Compte-tenu du codage des données choisi, nous devons définir les opérateurs de croisement et de mutation.

L'opérateur de croisement choisi (voir figure 21.2) est le suivant :

- la variable t_0 de l'avion 1 de l'enfant 1 est le barycentre de la variable t_0 de l'avion 1 du parent X affectée du coefficient α et de la variable t_0 de l'avion 1 du parent Y affectée du coefficient $(1-\alpha)$; pour l'enfant 2, on utilise les coefficients $(1-\alpha)$ et α ; ce processus est répété pour la variable t_0 de l'avion 2 et pour les variables t_1 des avions 1 et 2 des deux enfants ; à chaque fois, une valeur différente de α est choisie aléatoirement dans l'intervalle $[-0.5, 1.5]$[11] avec une probabilité uniforme ;
- la variable a de l'avion 1 de l'enfant 1 est choisie aléatoirement avec une probabilité $\frac{1}{2}$ parmi les variables a des avions 1 des deux parents.

L'opérateur de mutation choisi aléatoirement avec des probabilités identiques la variable t_0, t_1 ou a de l'avion 1 ou 2 d'un élément de population et le modifie en lui ajoutant un bruit gaussien (s'il s'agit de t_0 ou t_1) ou en lui ajoutant ou retranchant 10 degrés selon les possibilités (s'il s'agit de a).

La figure 21.3 montre une solution obtenue après 28 générations de l'algorithme génétique. Les paramètres choisis étaient :

---

11 Ne pas prendre α dans [0, 1] mais dans [−0.5, 1.5] permet d'éviter de se restreindre au segment dont les extrémités sont les variables des deux parents.

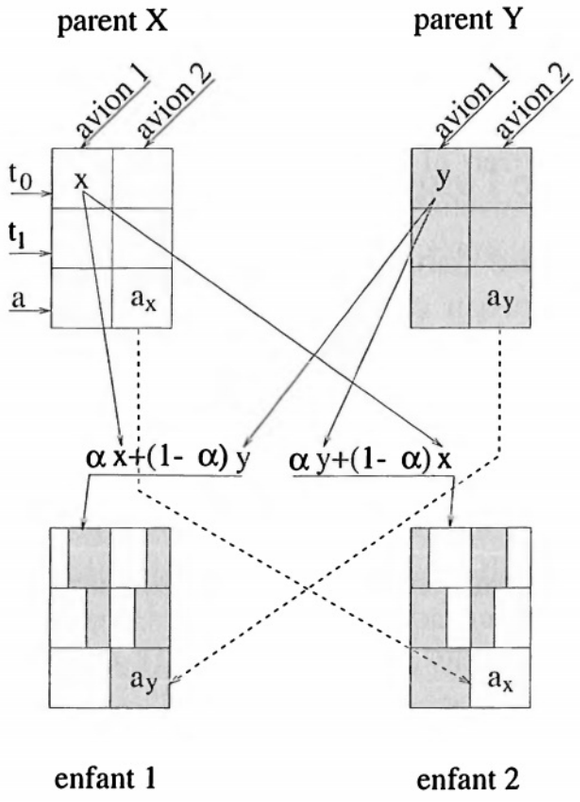


Figure 21.2 – Opérateur de croisement

- population comprenant 100 individus ;
- 60% de croisements à chaque génération ;
- 0.15% de mutations à chaque génération.

La durée d'une génération est alors de l'ordre de 3 centièmes de seconde sur un Pentium II 450. L'optimisation complète dure moins d'une seconde.

Cette solution est en fait optimale (ce résultat ne pouvant être assuré *a priori*, mais seulement constaté a posteriori). Un algorithme de type $A*$, appliqué au problème, fournit un résultat équivalent en un temps équivalent. L'algorithme génétique est donc, *pour cet exemple*, équivalent à une méthode de parcours de graphe.

21.4.4 Conflit à 5 avions

Nous avons ensuite voulu tester l'algorithme dans le cas où le conflit implique 5 avions initialement situés régulièrement sur un demi-cercle de 150 km de rayon et convergent vers le centre du cercle. La taille de la population a été augmentée à 500 individus. Les autres paramètres sont restés inchangés.

La figure 21.4 montre une solution obtenue après 200 générations de l'algorithme génétique. Le temps de calcul est de 1mn 20s.

La solution obtenue n'est pas optimale. Nous verrons par la suite comment on peut l'améliorer en utilisant des méthodes de *sharing* (section 21.5.5) et la *séparabilité partielle* du problème (section 21.5.6).

L'algorithme $A*$ ne permet pas de trouver la solution optimale car son temps d'exécution est trop long. 20mn de vol représentent 120 pas de temps de 10s. Un avion qui n'a pas encore engagé de manœuvre a 7 possibilités de modification de cap à chaque pas de temps (de -30 à 30 degrés par pas de 10). S'il a engagé une manœuvre, il peut soit la poursuivre, soit prendre un cap vers sa destination. Une fois sur la branche de retour, il n'a

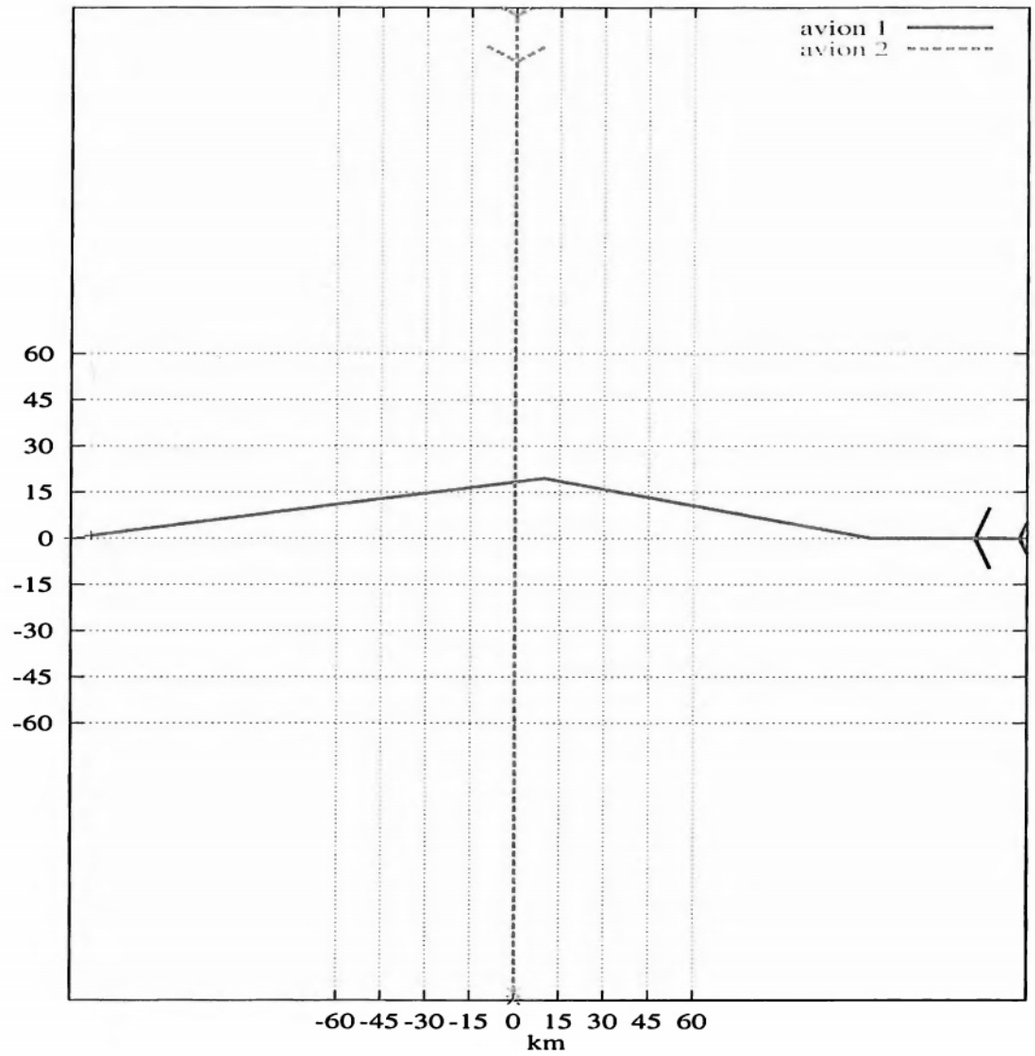


Figure 21.3 – Résultat de l'algorithme génétique (n=28)

plus d'alternative. La taille de l'arbre de recherche pour chaque avion est $7 + 3\,n\,(n+3)$ avec $n = 120$. Si le conflit implique p avions, la taille de l'arbre devient $[7+3\,n\,(n+3)]^p$. Pour 2 avions, la taille reste inférieure à 10^{10}. Pour 5 avions, on dépasse 10^{23}.

21.5 Techniques avancées

Nous allons examiner rapidement dans cette section différentes techniques complémentaires qui peuvent améliorer les performances des algorithmes génétiques. Ces techniques sont pour la plupart présentées en détail dans (Goldberg 1989c), à l'exception des représentations spécialisées, que l'on peut trouver dans (Michalewicz 1992), avec bien d'autres.

21.5.1 Utilisation d'un codage et d'opérateurs spécialisés

Les techniques de codage sous forme de chaînes de bits et de croisement standard présentent un certain nombre d'inconvénients :

- il n'est pas facile de choisir un « bon » codage adapté à la structure du problème ;
- l'application de la fonction de décodage lors de l'évaluation de la fitness est très coûteuse en temps de calcul ;

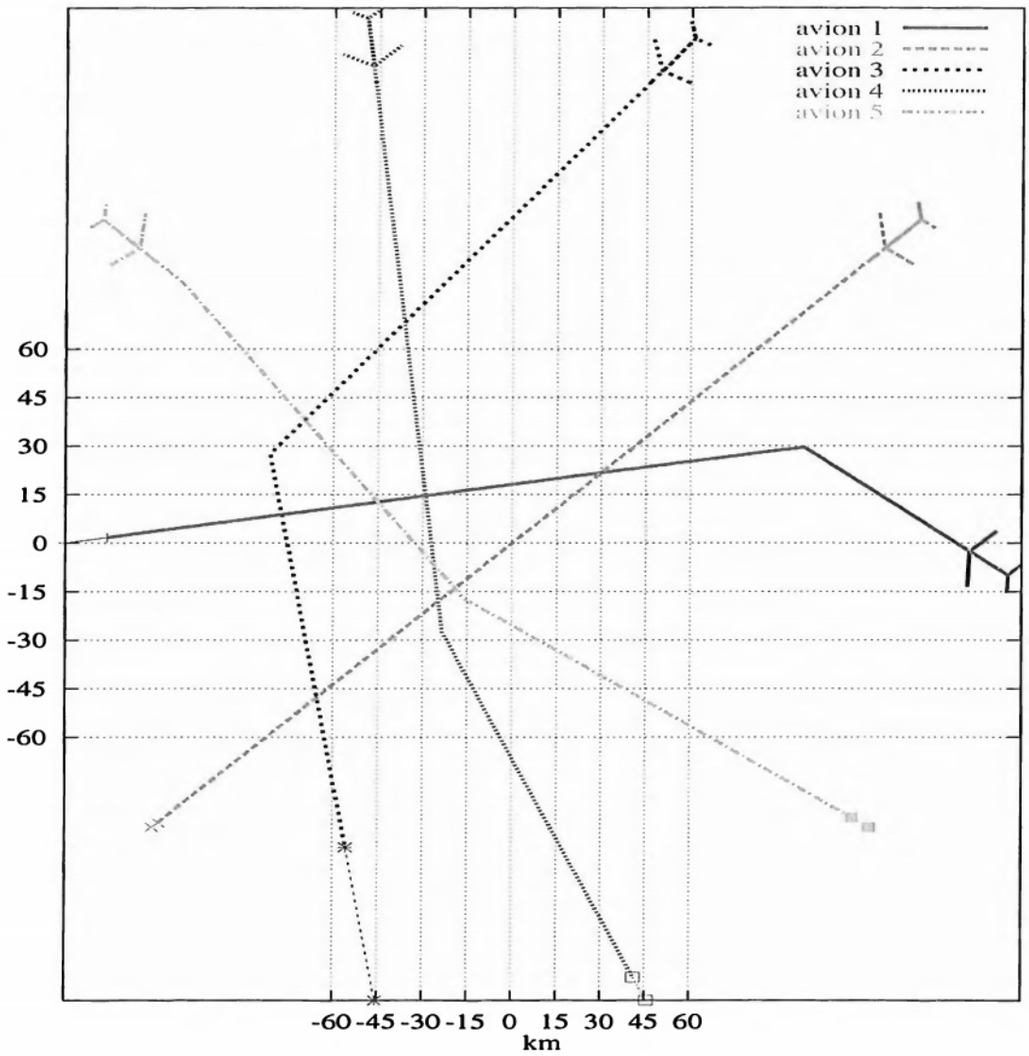


Figure 21.4 – Conflit à 5 avions

– les opérateurs de croisement/mutation sont « aveugles », et ne tiennent aucun compte de la structure du problème.

Il est alors naturel de tenter de définir des codages et des opérateurs spécialisés, qui s'appliqueront efficacement sur le problème considéré.

Dans l'exemple de la section 21.2.2, le codage en chaîne de bits a été abandonné au profit d'un codage utilisant directement les variables du problème. Les opérateurs de croisement et de mutations ont du alors être définis en fonction du codage. Les variables réelles et entières peuvent être traitées différemment. Ce type de représentation a pour but de mieux respecter la topologie de l'ensemble sur lequel l'optimisation est opérée à travers le codage.

Dans le cas des variables réelles, on emploie de façon classique le *croisement barycentrique*. Pour générer deux enfants, on commence par tirer un nombre α au hasard dans l'intervalle $[-0.5, 1.5]$ et par appliquer les formules suivantes :

$$\begin{aligned} p'_1 &= \alpha p_1 + (1-\alpha)p_2 \\ p'_2 &= \alpha p_2 + (1-\alpha)p_1 \end{aligned}$$

En ce qui concerne la mutation, on génère l'élément muté en appliquant la formule :

$$p'_1 = p_1 + B(0, \sigma)$$

où $B(0, \sigma)$ est un bruit gaussien centré en 0 et d'écart type σ. Le choix de σ dépend, bien entendu, du problème considéré.

Il est extrêmement important de réaliser que le codage et les opérateurs choisis conditionnent fortement la qualité de la solution obtenue. Les algorithmes génétiques ne sont pas des boîtes noires miraculeuses dans lesquelles il suffirait d'introduire une fonction pour en retirer l'optimum. Sans une réflexion appropriée sur le codage et les opérateurs, les résultats seront au mieux médiocres.

21.5.2 Opérateurs de réorganisation

L'un des problèmes des algorithmes génétiques est que le sens d'un allèle est lié à sa position, position qui dépend du codage. Ainsi, supposons que dans un premier codage une propriété importante soit codée (sur 2 bits) par le premier et le huitième bit d'une chaîne de 8 bits. Avec cette forme de codage, tout croisement opérera sur cette propriété. Si au contraire la propriété était codée sur le premier et le second bit (par exemple), le bloc serait presque insensible au croisement. C'est de cette constatation que vient l'idée de réorganiser les bits dans un individu, de façon à créer des blocs stables.

Un opérateur de réorganisation opère de la façon suivante : soit l'élément de population 10100010; on tire au hasard deux positions, par exemple 3 et 7. Tous les bits compris entre les positions 3 et 7, c'est à dire la séquence 1000, sont alors inversés, pour donner l'individu 10000110.

Se pose alors un premier problème : dans la présentation standard des algorithmes génétiques, le codage d'une propriété dépend de la position de chaque bit. En un mot, l'individu 10100010 et l'individu 10000110 ne code plus le même objet. Il faut donc rendre le décodage indépendant de la position en stockant avec la valeur de chaque bit la valeur de sa position. Ainsi, le premier individu deviendrait $(1,1),(2,0),(3,1),(4,0),(5,0),(6,0),(7,1),(8,0)$ et le second serait alors $(1,1),(2,0),(7,1),(6,0),(5,0),(4,0),(3,1),(8,0)$. Dans ces conditions, les deux individus représentent bien le même objet et seront décodés de la même façon.

Il reste un dernier problème à régler : comment effectuer un croisement ? En effet, considérons les deux individus :

$$(1,1),(2,0),(7,1),(6,0),(5,0),(4,0),(3,1),(8,0)$$

et

$$(1,1),(2,0),(3,1),(4,0),(5,0),(6,1),(7,1),(8,0)$$

Supposons que le site de croisement tiré soit le milieu. Nous aurions alors les deux individus :

$$(1,1),(2,0),(7,1),(6,0),(5,0),(6,1),(7,1),(8,0)$$

et

$$(1,1),(2,0),(3,1),(4,0),(5,0),(4,0),(3,1),(8,0)$$

Aucun de ces individus n'est viable, puisque le premier a deux fois les positions 6 et 7 et qu'il lui manque les positions 3 et 4 et réciproquement pour le second. Il faut donc définir différemment l'opérateur de croisement. Il existe diverses stratégies (voir (Goldberg 1989c)). Nous décrirons brièvement la technique dite de *partially matched crossover*

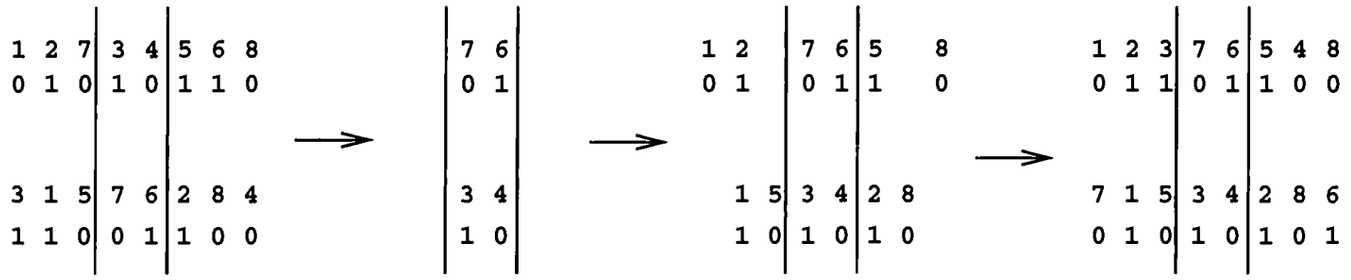


Figure 21.5 – Technique PMX

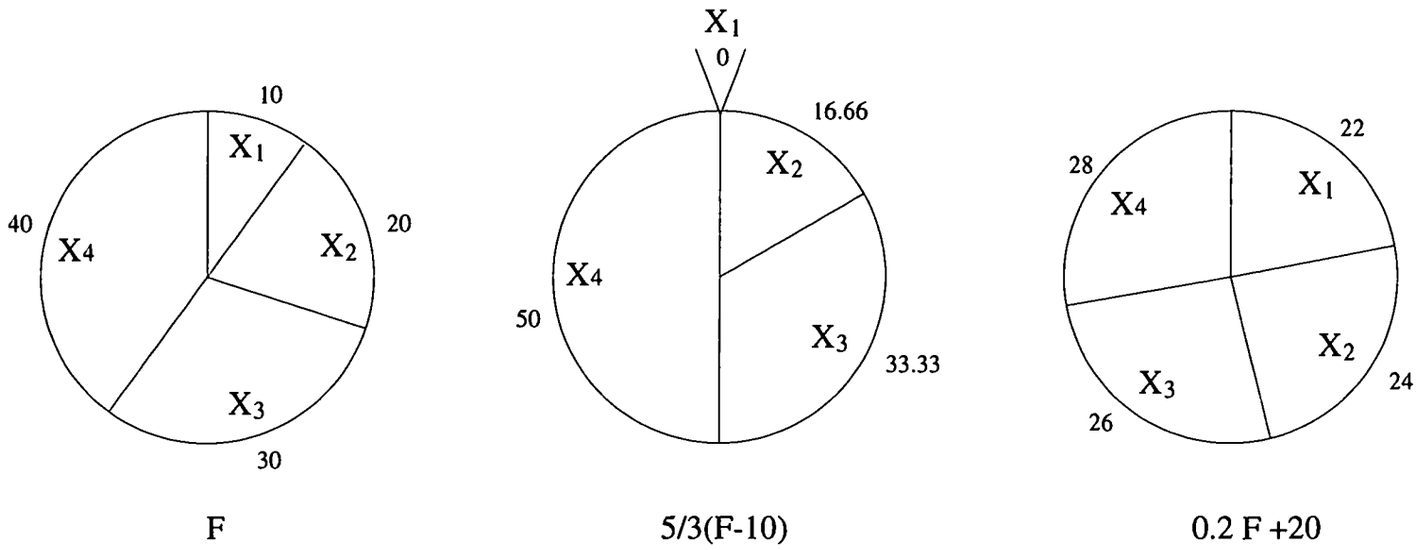


Figure 21.6 – Exemple de scaling

(PMX). Dans cette technique, on tire deux sites de croisement au lieu d'un seul (voir figure 21.5).

La zone située entre ces deux sites est d'abord échangée : on dit alors que l'allèle 3 *matche* (ou correspond à) l'allèle 7 et que l'allèle 4 matche l'allèle 6 (et réciproquement). Puis, on remet à leur place les allèles qui ne sont pas dupliqués après l'échange des zones de croisement. Dans le cas du premier individu, il s'agit des allèles autres que 7 et 6, et dans le cas du second, des allèles autres que 3 et 4. Puis on remplace dans l'individu du haut ce qui était l'allèle numéro 7 par son correspondant en numéro, c'est à dire l'allèle 3 de l'individu du bas ; on fait de même pour l'allèle 6, et on procède de la même façon pour les allèle 3 et 4 de l'individu du bas.

21.5.3 Le scaling ou mise à l'échelle

Le processus de sélection des individus qui consiste à reproduire en plus grand nombre les individus bien adaptés que les individus mal adaptés conditionne fortement la convergence de l'algorithme. La technique de mise à l'échelle ou *scaling* a pour but de modifier la pression de sélection. Elle consiste à composer la fonction d'évaluation avec une fonction croissante. Considérons l'exemple suivant où les 4 individus X_1, X_2, X_3 et X_4 de la population ont pour adaptation 10, 20, 30 et 40. Si l'on applique les fonctions linéaires croissantes $f \rightarrow \frac{5}{3}(f - 10)$ ou $f \rightarrow 0.2\,f + 20$ sur la valeur de l'adaptation, on donne plus ou moins d'importance aux individus (voir figure 21.6). Dans le premier cas, on augmente la pression de sélection, dans le deuxième cas, on la diminue.

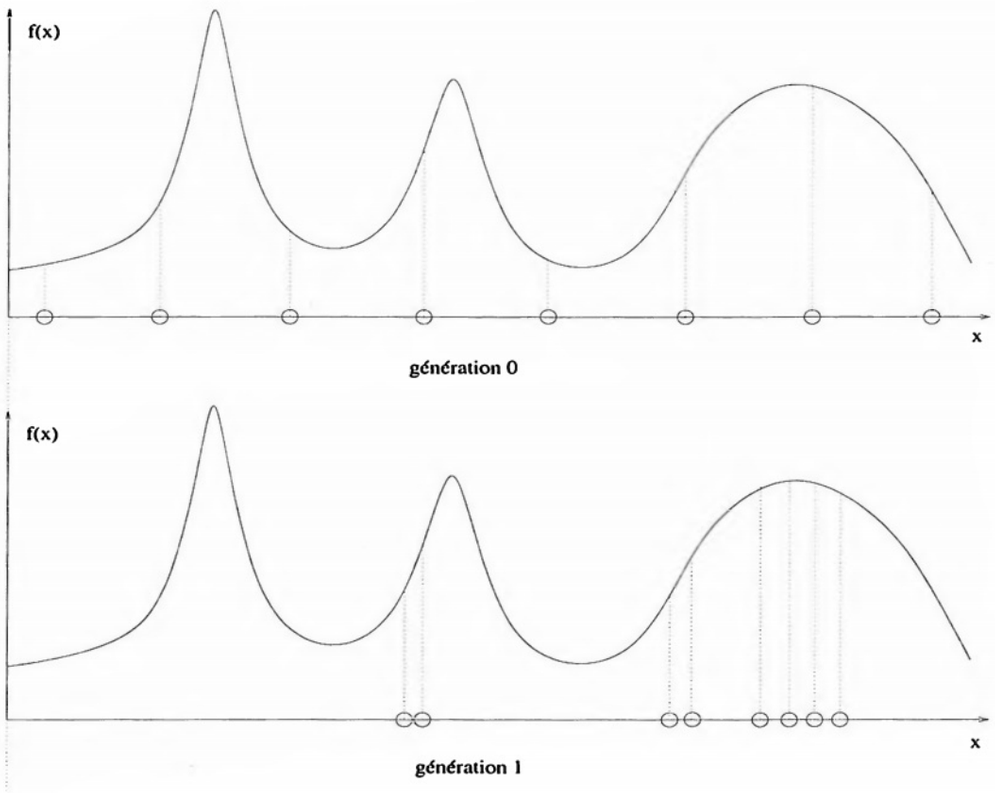


Figure 21.7 – Intérêt du sharing

21.5.4 Modes de sélection, tournoi

Le mode de sélection des individus présenté dans la section 21.2.2 se contente de reproduire les individus proportionnellement à leur critère d'adaptation. D'autres modes de sélection sont envisageables, comme la sélection par le rang (les individus sont classés dans la population, et leur adaptation n'est plus fonction que du rang qu'ils occupent). On peut également envisager, lors de l'exécution de l'opérateur de croisement, de conserver parmi les deux parents et les deux enfants, les deux individus qui ont la meilleure adaptation. On effectue alors un tournoi entre les parents et les enfants à l'issue du croisement. Le danger de ce principe est la convergence prématurée de la population vers un minimum local.

21.5.5 Le sharing

Un des problèmes souvent rencontrés dans l'utilisation des algorithmes génétiques est la disparition trop rapide de certains éléments de la population en raison de la surdomination d'un individu particulier, dont les représentants finissent par constituer la majeure partie de la population. Considérons l'exemple de la figure 21.7 : à la génération 0, la population de 8 individus est uniformément répartie sur l'intervalle de recherche. Malheureusement, les meilleurs individus ayant une plus forte probabilité de reproduction que les mauvais, on peut se retrouver, à la génération 1 avec une population répartie autour de minima locaux. Le but du sharing est de contrecarrer ce phénomène.

De plus, un des avantages des algorithmes génétiques par rapport aux algorithmes de type recuit simulé est de présenter plusieurs solutions viables, et non une seule, au terme d'une exécution du programme. Il faut donc éviter l'élimination de tous les individus au bénéfice d'un seul, même s'il est mieux adapté.

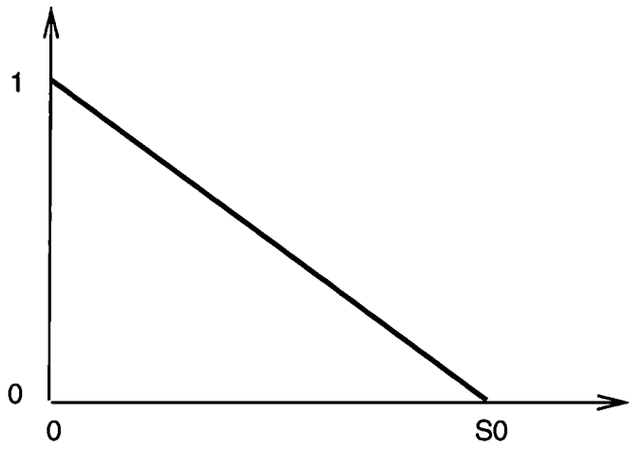


Figure 21.8 – Fonction de sharing

La technique dite de *sharing* (que l'on pourrait traduire par *partage* ou *répartition* en français) tente de faire disparaître ce problème. Le principe est simple : la fitness de chaque individu est modifiée en fonction du nombre d'individus qui l'entoure dans l'espace du problème ; plus ce nombre est grand, et plus on diminue la fitness. La formule classiquement utilisée est :

$$f_s(x_i) = \frac{f(x_i)}{\sum_{j=1}^{n} s(d(x_i, x_j))}$$

$f_s(x_i)$ est la valeur modifiée de la fitness de l'élément x_i après l'application de l'opération de sharing ; $f(x_i)$ est la fitness « traditionnelle » ; $d(x_i, x_j)$ est la distance séparant les éléments x_i et x_j ; s est une fonction « astucieusement » choisie. Cette fonction a souvent la forme triangulaire représentée sur la figure 21.8.

On voit que tout élément situé à une distance supérieure à s_0 ne modifie en rien la fitness, mais qu'en revanche, plus il y a d'éléments proches et plus la fitness se trouve diminuée. Cette technique permet ainsi de pénaliser les zones trop peuplées, contenant trop de représentants d'une même solution, ou d'une solution proche ; elle favorise en revanche les représentants uniques ou peu nombreux d'une solution.

Le sharing nécessite la définition d'une distance sur l'espace de recherche et pénalise le temps d'exécution de l'algorithme car si la population est de taille n, le temps de calcul des distances entre les paires d'individus est proportionnel à n^2. Il existe des méthodes approchées (Yin and Germay 1993) ou permettant de ramener le temps de calcul proportionnel à $n\,log(n)$.

L'utilisation du sharing sur l'exemple de résolution de conflit permet d'une part d'améliorer la solution obtenue (figure 21.9) et d'autre part d'obtenir plusieurs solutions différentes.

21.5.6 Fonctions partiellement séparables : croisement adapté

Les algorithmes génétiques sont très souvent utilisés pour des problèmes de grande taille où l'on cherche à minimiser un critère additif, somme de critères faisant chacun intervenir une partie des variables. Ce type de fonction n'est pas marginal, beaucoup de problèmes pratiques faisant intervenir un grand nombre de variables sont partiellement

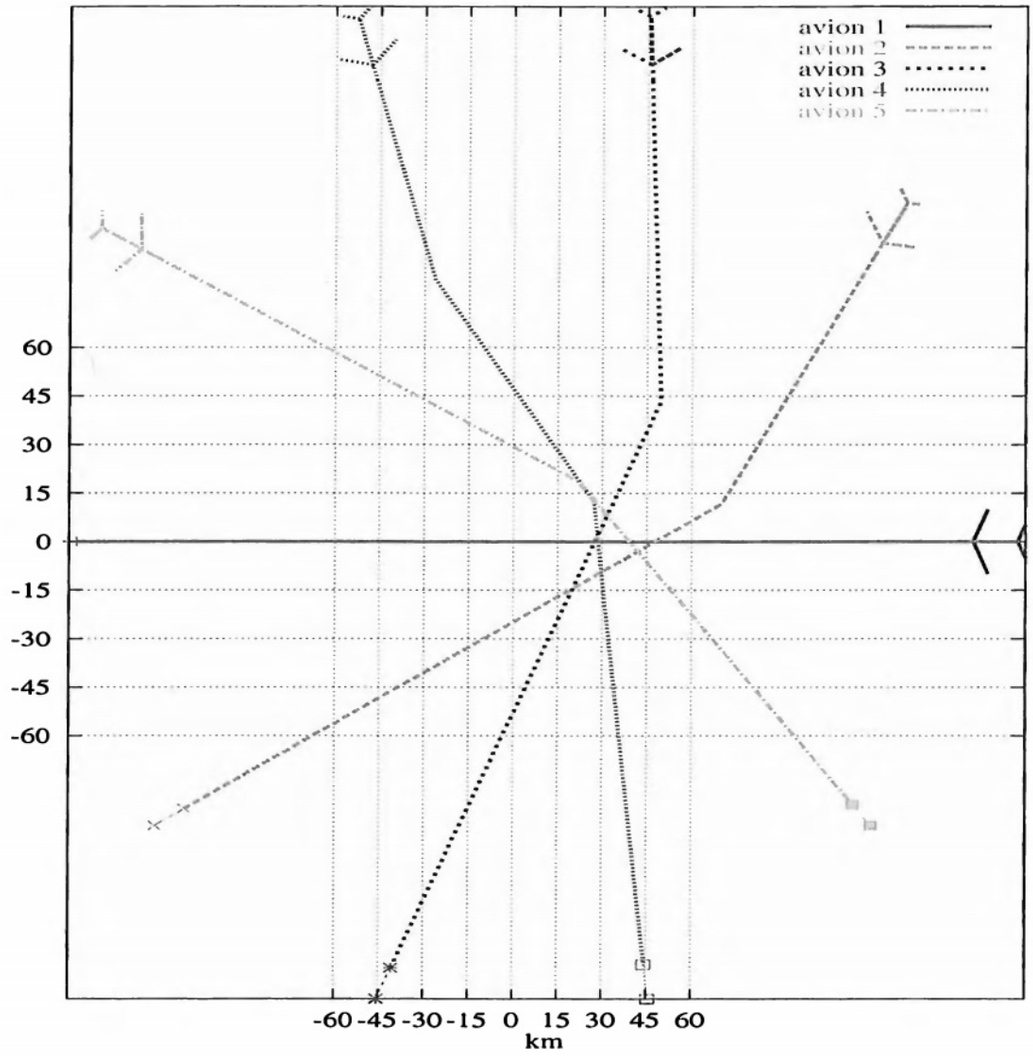


Figure 21.9 – Conflit à 5 avions avec sharing

séparables. Nous verrons à la fin de cette section que notre problème de résolution de conflits aériens en fait partie.

L'idéal, pour résoudre un problème à plusieurs variables est de pouvoir séparer le problème en sous-problèmes indépendants. La solution globale est reconstituée à partir des solutions de chacun des sous-problèmes. Considérons l'exemple de la fonction définie sur $\Re^n$ $(x_1, .., x_n) \rightarrow \sum_1^n x_i^2$. Cette fonction est complètement séparable, elle est la somme de n critères positifs $x_i \rightarrow x_i^2$ que l'on peut minimiser indépendamment pour trouver la solution globale. Supposons que l'on n'ait pas profité de cette propriété et considérons l'opérateur de croisement simple d'un AG qui, à partir de deux parents $(x_1, .., x_n)$ et $(y_1, .., y_n)$ génère les enfants $(z_1, .., z_n)$ et $(\bar{z}_1, .., \bar{z}_n)$ avec $(z_i, \bar{z}_i) = (x_i, y_i)$ ou (y_i, x_i) . Or le meilleur enfant que l'on peut produire est obtenu en choisissant pour z_i la variable x_i ou y_i qui minimise le critère $t \rightarrow t^2$.

On peut ainsi largement améliorer la convergence d'un algorithme génétique en utilisant un opérateur de croisement qui tient compte de la séparabilité du critère à optimiser pour fabriquer les enfants à partir de 2 parents.

Par la suite, on appellera fonction partiellement séparable toute fonction positive F de n variables x_1, x_2, .., x_n pouvant s'écrire comme la somme de m fonctions positives

F_i, chacune ne faisant intervenir que n_i variables ($n_i < n$).

$$F(x_1, x_2, .., x_n) = \sum_{i=1}^{m} F_i(x_{j_1}, x_{j_2}, .., x_{j_{n_i}})$$

On définit l'adaptation ou *fitness locale* $G_k(x_1, x_2, .., x_n)$ pour la variable x_k comme suit :

$$G_k(x_1, x_2, .., x_n) = \sum_{i \in S_k} \frac{F_i(x_{j_1}, x_{j_2}, .., x_{j_{n_i}})}{n_i}$$

où S_k est l'ensemble des i tels que x_k est une variable de F_i et n_i le nombre de variables de F_i.

Intuitivement, l'adaptation locale d'une variable isole sa contribution à l'adaptation globale. On a de plus la propriété suivante :

$$\sum_{k=1}^{n} G_k(x_1, x_2, .., x_n) = F(x_1, x_2, .., x_n)$$

L'opérateur de croisement fabrique un enfant $z_1, .., z_n$ à partir de deux parents comme suit :

- si $G_k(x_1, .., x_n) < G_k(y_1, ..y_n) - \Delta$, alors $z_i = x_i$;
- si $G_k(x_1, .., x_n) > G_k(y_1, .., y_n) + \Delta$, alors $z_i = y_i$;
- si $|G_k(x_1, .., x_n) - G_k(y_1, .., y_n)| \leq \Delta$, alors $z_i = x_i$ ou y_i avec une probabilité $\frac{1}{2}$ ou peut être une combinaison linéaire des deux variables s'il s'agit de variables réelles.

Le paramètre Δ rend plus où moins aléatoire la construction de l'enfant.

Ce principe s'applique très bien au problème de résolution de conflits : en effet, on cherche d'abord à minimiser C somme de critères faisant intervenir des paires d'avions, puis on cherche à minimiser la somme des allongements, critères ne faisant intervenir qu'un avion chacun. Le croisement adapté permet, pour le problème de résolution de conflit, de résoudre des configurations atteignant une vingtaine d'avions, comme le montre la figure 21.10.

21.5.7 Structures de données complexes

Les algorithmes génétiques peuvent optimiser n'importe quel type de structure de données dès lors que l'on est capable de définir une fonction d'évaluation et des opérateurs de recombinaison. On peut ainsi, comme le décrit l'exemple 21.5.7 optimiser les poids d'un réseau de neurone, voire même sa structure. Pour un problème d'optimisation opérant sur des vecteurs réels, on peut définir comme élément de population non plus le vecteur lui-même, mais un simplexe de vecteurs (une méthode locale permettant d'optimiser le simplexe à chaque génération (Durand and Alliot 1999)). On peut également envisager des éléments de population diploïdes.

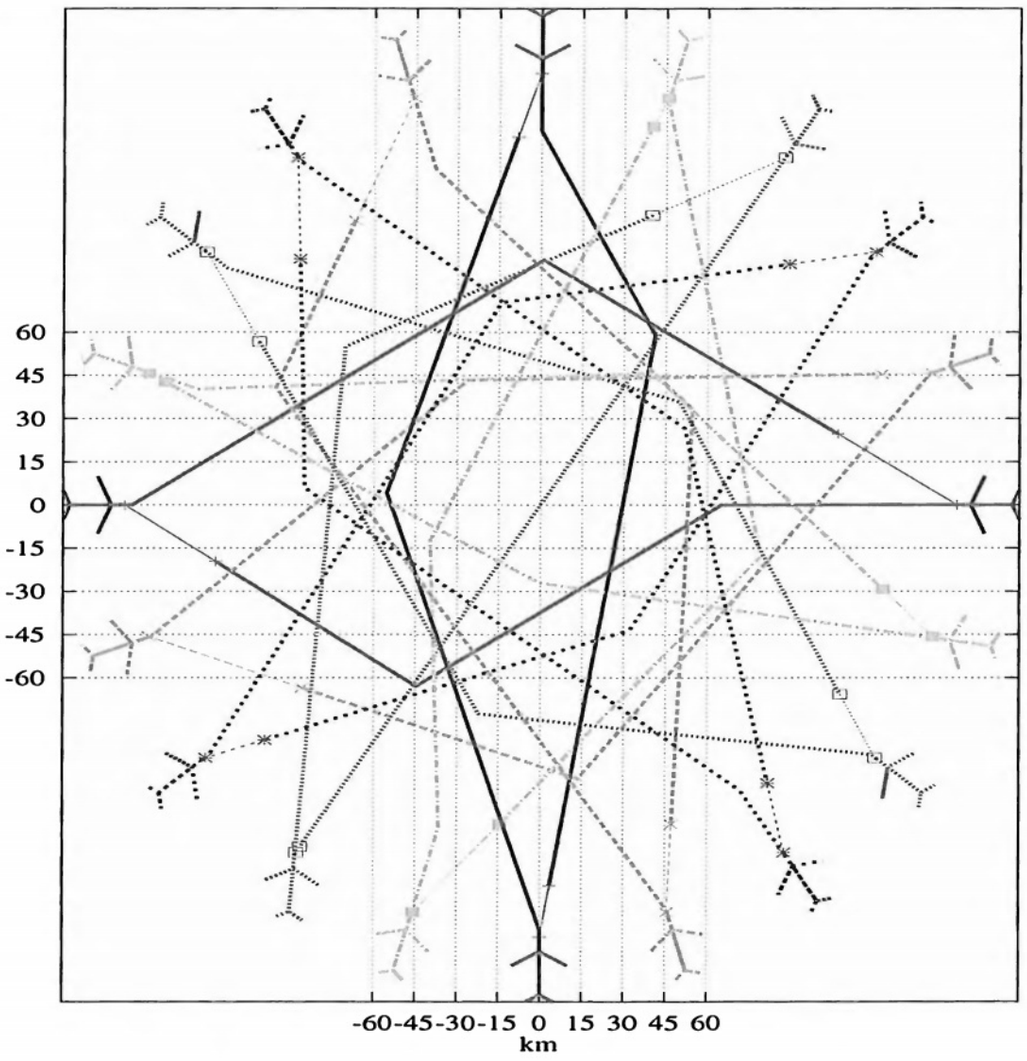


Figure 21.10 – Conflit à 20 avions

Diploïdie et dominance

Il était inévitable que l'on tente d'utiliser pour les algorithmes génétiques des chromosomes diploïdes en lieu et place des chromosomes haploïdes utilisés jusqu'ici : les organismes vivants les plus évolués ont des chromosomes diploïdes[12]. Dans ce cas, le décodage du génotype au phénotype se fait en introduisant la notion de dominance d'un gène sur un autre, comme en génétique classique (voir (Goldberg 1989c)).

Les diverses expériences faites sur la diploïdie tendent à montrer que cette technique n'est efficace que lorsque la fonction fitness varie en fonction du nombre de générations de l'algorithme génétique *i.e.*, f est de la forme $f(C, n)$ (où n est le numéro de génération de l'algorithme génétique), au lieu d'être simplement de la forme $f(C)$. Ceci semble être en accord avec la génétique qui pense qu'un des facteurs positifs de la diploïdie est sa capacité à mémoriser (par l'intermédiaire de gènes récessifs) des caractéristiques qui ont pu se révéler utiles dans le passé, ne le sont plus aujourd'hui, et donc doivent disparaître du phénotype ; cependant, si une évolution de l'environnement venait à rendre ces caractéristiques utiles à nouveau, les gènes récessifs les codant étant disponibles, elles réapparaîtraient sur le champ.

12 Un chromosome haploïde est un chromosome à un seul brin (une seule chaîne de bits), un chromosome diploïde est un chromosome à deux brins (deux chaînes de bits).

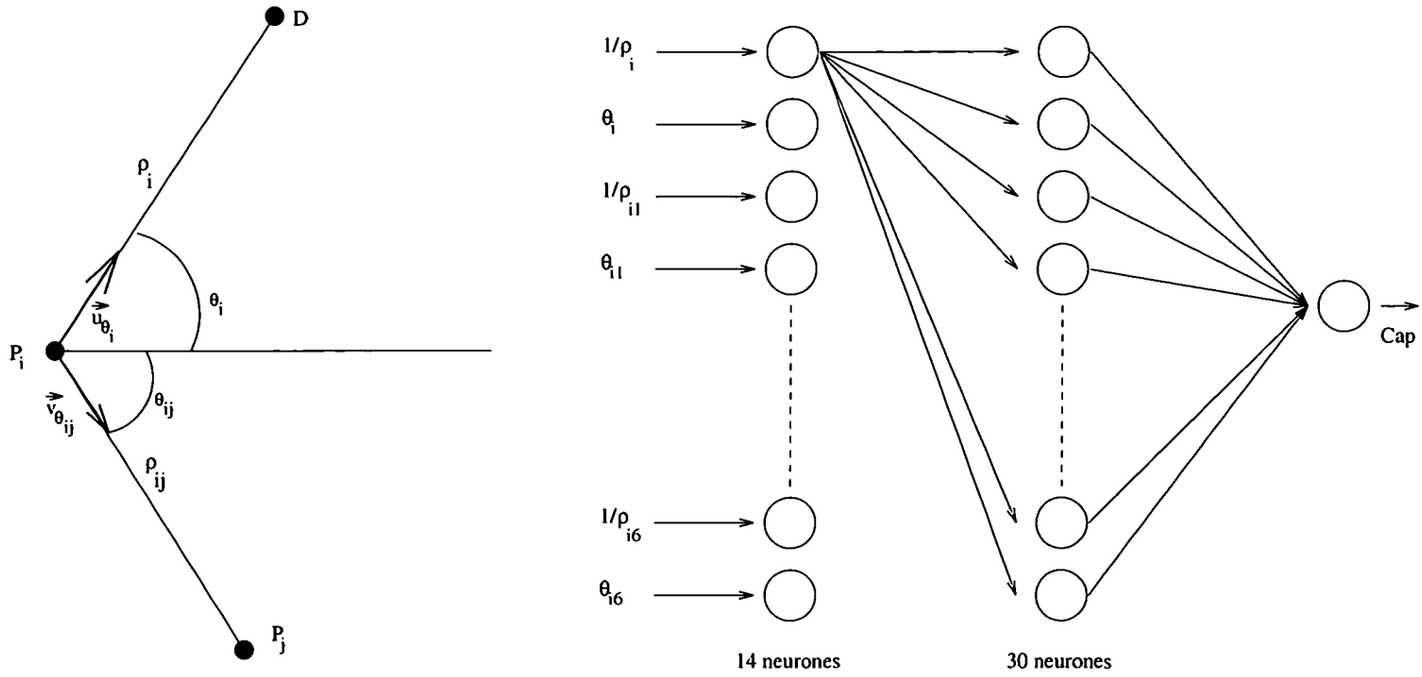


Figure 21.11 – Contrôle de mobiles par réseau de neurones

La diploïdie est, pour l'instant, peu utilisée pour la programmation des algorithmes génétiques.

Algorithmes génétiques et réseaux de neurones

Nous allons présenter dans cette section une intéressante combinaison reliant les algorithmes génétiques aux réseaux de neurones.

Considérons le problème suivant : nous disposons dans un espace clos de six mobiles. Chacun de ses mobiles à une origine et une destination. Le but de chacun de ces engins est de rejoindre sa destination en suivant le plus court chemin. Cependant, ils ne doivent à aucun moment se trouver à une distance d'un autre mobile inférieure à une norme de séparation D, et ce durant l'intégralité de leur déplacement ; il s'agit donc de construire une trajectoire optimale au sens de la distance tout en respectant des contraintes de séparation.

Pour chaque mobile i, les paramètres en entrée sont les distances ρ_{ij} aux autres mobiles, les gisements de ces mobiles θ_{ij} ainsi que la distance ρ_i et le gisement θ_i de sa destination. En sortie, le programme doit fournir un cap à suivre α_i pour le pas de temps suivant (on suppose que les mobiles avancent par pas de temps discret, à une vitesse unique et fixée).

On peut donc construire un réseau de neurones qui aurait pour entrées les différents paramètres et fournirait en sortie le cap du mobile (voir figure 21.11). Le problème est de réaliser l'apprentissage du réseau. On ne dispose en effet d'aucune base d'exemple permettant de faire de l'apprentissage supervisé, car il n'existe pas de solution mathématique générale à ce problème.

On peut alors utiliser la technique des algorithmes génétiques pour calculer les poids du réseau de neurones. On commence par générer un ensemble de poids pour notre réseau (un chromosome contient en fait la matrice des poids). Puis, on fait se déplacer nos mobiles, sur un exemple ou un ensemble d'exemple, avec les poids de chacun des chromosomes. La *fitness* de chaque chromosome correspond à la qualité de la trajectoire

générée. Le plus court chemin est bien sur l'optimal, mais il faut éviter les autres mobiles. Toute collision pénalise la fitness, soit en l'annulant, soit en la divisant par un coefficient donné.

Les opérateurs de croisement sont du type barycentrique. On tire aléatoirement deux coefficients dans deux réseaux et on réalise une combinaison linéaire pour obtenir les nouveaux coefficients. La mutation consiste à ajouter un bruit blanc sur un coefficient tiré au hasard.

On peut ainsi construire un réseau qui parviendra à réaliser les deux objectifs fixés : de bonnes trajectoires en terme de distance et un évitement des autres mobiles[13].

La combinaison des algorithmes génétiques et des réseaux de neurones ouvre une voie particulièrement intéressante pour les problèmes où l'on ne dispose pas de base de données d'exemples permettant de faire de l'apprentissage supervisé.

21.5.8 Recherche multi-objectifs

Il est rare, pour un problème réel, que l'on n'ait qu'un seul critère à optimiser ; bien souvent, on se contente de rechercher un compromis entre plusieurs critères (par exemple une combinaison linéaire entre les critères). Or les algorithmes génétiques font évoluer une population d'individus, et on peut être tenté de faire converger cette population non plus vers un point optimum, mais vers la surface de Pareto des différents critères.

Dominance au sens de Pareto

La notion de dominance au sens de Pareto se définit comme suit : on dit que le point A domine le point B si, $\forall i$, $f_i(A) \geq f_i(B)$ (avec au moins une inégalité stricte), où les f_i représentent les critères à maximiser. L'ensemble des points qui ne sont dominés par aucun autre point forme la surface de Pareto. Tout point de la surface de Pareto est "optimal", dans la mesure où on ne peut améliorer la valeur d'un critère pour ce point sans diminuer la valeur d'au moins un autre critère.

On peut construire un algorithme génétique permettant de trouver la surface de Pareto d'un problème multi-critères.

Modification de l'opérateur de sélection

La technique employée dérive directement des travaux de Jeffrey Horn et Nicholas Nafpliotis ((Horn and Nafpliotis 1993)). Au lieu de travailler sur une fonction d'évaluation scalaire, on a désormais une fonction d'évaluation vectorielle (chaque critère est une variable du vecteur). En conséquence, le principal changement induit concerne le processus de sélection[14].

On introduit une variante de la notion de dominance que l'on définit ainsi : l'élément E_i domine E_j si le nombre des valeurs contenues dans son vecteur d'adaptation qui sont supérieures aux valeurs correspondantes dans E_j dépasse un certain seuil. La technique proposée pour effectuer la sélection est alors simple : on tire deux individus au hasard

---

13 Ce travail s'inspire d'un travail similaire (Schoenauer *et al.* 1993).

14 En ce qui concerne les autres opérateurs de base, à savoir le croisement et la mutation, il n'y a aucune modification, car ceux-ci travaillent dans l'espace d'état et non dans l'espace résultat.

ainsi qu'une sous-population[15] à laquelle ils n'appartiennent pas et qui va servir à les comparer.

Trois cas se présentent alors :
- si le premier élément domine tous ceux de la sous-population, et que ce n'est pas le cas pour le second, alors le premier sera sélectionné.
- inversement, si seul le second élément domine l'ensemble de la sous-population, alors c'est lui qui sera conservé.
- il reste deux possibilités : soit les deux sont dominés, soit les deux dominent. On ne peut se prononcer sans ajouter un autre test, c'est pourquoi dans ce cas il est fait usage d'un nouveau type de sharing qui opère sur l'espace objectif.

Le sharing va conduire à sélectionner celui des deux individus qui a le moins de voisins proches, autrement dit, on élimine celui qui se situe dans une zone d'agrégation pour conserver celui qui est dans une région moins dense.

Le terme *voisin proche* n'a pas de signification précise mais il est possible de définir un voisinage (aussi appelé niche) correct en se servant de la distance de Holder :

$$d_H(E_i, E_j) = \left(\sum_{k=1}^{n} |f_i^k - f_j^k|^p \right)^{\frac{1}{p}}$$

f_i^k désignant la k-ième composante du vecteur adaptation de l'élément i. Le paramètre p permet de faire varier la forme et la taille du voisinage. A l'intérieur des voisinages ainsi définis dans l'espace objectif, il suffit de compter le nombre d'individus pour favoriser les zones les moins denses, et de cette façon, maintenir la diversité de la population.

21.5.9 Algorithmes génétiques et coévolution

Pour certains problèmes d'optimisation, la fonction fitness résulte d'une évaluation sur un certain nombre de cas tests. Or bien souvent, il est difficile de choisir une base de cas tests représentative de tous les cas possibles. Imaginons par exemple que l'on veuille approcher la courbe que nous fournit une boîte noire par un polynôme de degré n et que l'on veuille déterminer les coefficients du polynôme en utilisant un algorithme génétique. On peut sélectionner un certain nombre de points de la courbe qui permettront, pour un polynôme donné de mesurer la qualité du polynôme retenu. Rien ne garantit toutefois qu'une autre sélection de points ne donne pas une évaluation totalement différente.

Le principe de la coévolution est de faire évoluer deux populations différentes en parallèle. Dans le cas présent, la première population est une population d'individus codant les coefficients d'un polynôme, la deuxième population est une population de points de la courbe servant à tester la première population.

Il n'existe pas d'algorithme de coévolution *canonique*. Différents travaux ayant utilisé le principe de coévolution permettent de donner une image plus concrète de ce que peut être la coévolution :

---

15 La sous-population tirée aura une taille proportionnelle à celle de la population de départ (généralement autour de 10%). Seule une partie des individus est ainsi utilisée, ce qui permet de réduire le temps de calcul.

- W. D. Hillis a été un des premiers à utiliser des techniques de coévolution du type *proie-prédateur*. Ce nom provient du fait qu'il s'agit de faire évoluer simultanément, mais de manière concurrente, deux populations. Hillis emploie le mot de *parasites* pour désigner les éléments de la population concurrente, qui en exploitant les failles éventuelles des éléments de la première population, les poussent à s'améliorer.
  Dans un article de 1992, *Co-evolving Parasites Improve Simulated Evolution as an Optimization Procedure* (Hillis 1992), Hillis présente une méthode dont le but est de générer des réseaux de tri, parvenant à trier un nombre n donné de nombres au moyen du plus petit nombre possible de comparaisons successives. Les éléments de la première population sont des réseaux de tri, qui évoluent au moyen d'un algorithme particulier. Nous développerons les caractéristiques de cet algorithme qui nous semblent jouer un rôle important dans l'utilisation par Hillis de la coévolution. Les *parasites* sont des ensembles de *cas tests*, c'est-à-dire des séquences d'entiers (de 0 et de 1 puisque un réseau de tri qui trie correctement toute séquence de 0 et de 1 de taille n trie correctement toute séquences de nombres de taille n). Les réseaux ont une adaptation d'autant plus grande qu'ils parviennent à ordonner correctement une plus grande proportion de ces cas tests. La coévolution consiste dans ce cadre a faire évoluer, en même temps que la population de réseaux, une population de *parasites*.
  Hillis obtient grâce à la coévolution de meilleurs résultats que ceux qu'il avait obtenus au moyen de l'évolution simple des réseaux, tant en termes d'efficacité des réseaux pour un plus petit nombre de comparaisons, qu'en terme de rapidité d'obtention de ces réseaux. Il voit à ce résultats deux explications. D'abord, les réseaux imparfaits (qui ne trient pas correctement toutes les séquences de n entiers), qui pourraient constituer des optima locaux sont la proie de parasites adaptés à leurs faiblesses qui se développent dans leur voisinage, et ainsi ne sont pas viables (on voit ici le rôle de la localisation des éléments des deux populations sur une grille). De plus, les cas tests les plus difficiles dominent rapidement dans la population des parasites, ce qui évite des évaluations inutiles de réseaux sur des cas tests trop faciles.
- Jan Paredis (Paredis 1994; 1996b; 1996a) utilise aussi des méthodes de coévolution du type *prédateur-proie*. Il utilise conjointement un algorithme de coévolution (CGA : Co-evolutionary Genetic Algorithm), et une technique d'apprentissage, qu'il appelle *Life Time Fitness Evaluation* (LTFE). Il donne dans *coevolutionary computation* (Paredis 1996a) une présentation générale de son algorithme CGA tel qu'il peut être appliqué à ce qu'il appelle des *test-solution problems*. c'est-à-dire des problèmes dans lesquels on cherche à obtenir des solutions générales à appliquer à un grand nombre de cas différents, et pour lesquels on peut disposer d'un (grand) nombre de cas tests. La recherche de réseaux de tri décrite dans l'article de Hillis est un cas particulier de problème solution-test, et Jan Paredis développe dans les différents articles cités ici l'application de son algorithme à différents problèmes tels que la recherche de réseaux neuronaux de classification (classification neural networks) (Paredis 1996a), ou des problèmes de satisfaction de contraintes (CSP) (Paredis 1994; 1996a).

21.6 Stratégies d'évolution

Les stratégies d'évolution sont en fait généralement désignées, même dans les milieux francophones, par leur nom anglais de *Evolution Strategies*, ou par le sigle *ES*. Elles ont été introduites par Schwefel, à Berlin en 1965. à l'origine, un individu unique produisait, au moyen d'une mutation, un descendant unique, une sélection s'opérait entre eux deux.

Dès leurs origines, une grande originalité des stratégies d'évolution par rapport aux algorithmes génétiques a résidé dans leur utilisation du codage réel (voir section 21.5.1) et de la mutation gaussienne (section 21.5.1).

Elles ont été étudiées et développées par Rechenberg, à qui on doit la règle dite du un cinquième : le taux de mutation est modifié au cours du déroulement de l'algorithme de manière à atteindre un taux de succès des mutation de un cinquième, c'est-à-dire que dans un cas sur cinq en moyenne, le produit d'une mutation (le *fils*) a une meilleure *fitness* que l'élément à partir duquel il a été obtenu (le *père*). Des techniques similaires ont pu ensuite être reprises dans le cadre d'autres algorithmes évolutionnaires, de même que la *self adaptation*, introduite par Schwefel (voir section 21.5.1).

Dans les années 70 ont été développées 2 versions des stratégies d'évolution, les $(\mu + \lambda)$-ES et les (μ, λ)-ES, dues à Schwefel, dont on trouvera une description dans (Bäck and Schwefel 1993). Dans les deux cas, μ individus donnent, au moyen de mutations (et éventuellement d'opérateurs de recombinaisons, tels que le croisement décrit dans la section 21.5.1), λ nouveaux individus. Dans le cadre des (μ, λ)-ES, $\lambda > \mu$, et la sélection retient μ individus sur les λ qui ont été créés. Elle retient, dans le cadre des $(\mu + \lambda)$-ES, μ individus sur l'ensemble des μ *parents* et des λ *enfants*. La sélection est déterministe : les meilleurs λ individus (en termes de *fitness*) sont conservés, les λ moins bons sont éliminés.

Les stratégies d'évolution, en plus de former un groupe d'algorithmes très intéressant en eux mêmes, ont eu une influence positive sur développement des algorithmes génétiques, notamment par la mise au point de nouveaux opérateurs de mutation, et par l'expérience du codage réel qu'elles ont permis d'acquérir.

21.7 Programmation évolutionnaire

Le principe de la programmation évolutionnaire, ou Evolutionnary Programming a d'abord été présenté par L. J. Fogel (Fogel *et al.* 1966) en 1966.

Ces algorithmes ont été grandement développés (notamment par le passage au codage réel) et popularisés par D. B. Fogel (Fogel *et al.* 1992).

Leurs principales caractéristiques, par rapport aux autres algorithmes évolutionnaires, étaient, à l'origine, le fait qu'ils n'utilisaient pas de croisement, et surtout l'utilisation d'un mode de sélection originale, la sélection par tournoi.

Nous donnons ici une description simple d'un algorithme de programmation évolutionnaire (cette famille d'algorithmes s'est rapidement enrichie et diversifiée). μ individus en donnent μ par mutation. Si le codage utilisé est un codage réel, cette mutation est gaussienne, et son écart-type peut, au moment de la mutation d'un individu, être calculé à partir de la *fitness* de cet individu.

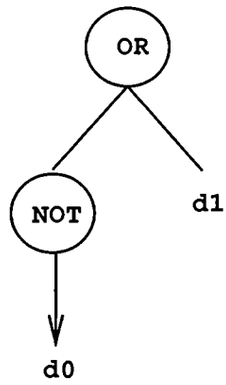


Figure 21.12 – Fonction LISP représentée par un arbre

Notons $\mathcal{P}(n)$ la population des μ individus présent à la génération n, et $\mathcal{P}'(n)$ les μ individus obtenus à partir d'eux par mutation. On va d'abord attribuer une note à chacun des 2 μ individus de $\mathcal{P}(n) \cup \mathcal{P}'(n)$. Pour chaque individu e de $\mathcal{P}(n) \cup \mathcal{P}'(n)$, on tire aléatoirement q individus dans $\mathcal{P}(n) \cup \mathcal{P}'(n)$. La note de e est alors égale au nombre d'individus (parmi ces q) dont la fitness est inférieure à celle de e.

La sélection retient pour former la nouvelle population $\mathcal{P}(n+1)$, les μ éléments ayant obtenu les meilleures notes.

Les performances de représentants de ces deux familles d'algorithmes (ES et EP), et d'un algorithme génétique (utilisant un codage binaire) sont comparées sur des problèmes d'optimisation de fonctions test de $\Re^n$ dans $\Re$, dans (Bäck and Schwefel 1993).

21.8 La programmation génétique

La programmation génétique est une technique récente qui cherche à appliquer les techniques génétiques sur des ensembles de programmes et non sur des données. Nous ne décrirons que très brièvement la programmation génétique, le lecteur intéressé devra se reporter à (Koza 1992) : une présentation exhaustive, accompagnée de nombreux exemples, des techniques de programmation génétique.

Prenons un exemple simple : supposons que nous souhaitions construire une fonction LISP à partir des fonctions logiques classiques (`AND`, `OR`, `NOT`) qui implante l'implication. Il est clair que dans ce cas, la fonction s'écrit simplement :

```
(define (implication d0 d1) (OR (NOT d0) d1))
```

La programmation génétique permet de générer automatiquement une telle fonction à partir de la table de vérité de l'implication. Le fonctionnement est le suivant :

1. on commence par générer une population originelle comprenant, par exemple, 200 programmes. Un programme est représenté par un arbre ne contenant que les nœuds `AND`, `OR` et `NOT`, et dont les terminaux sont `d0` ou `d1`. On peut voir sur la figure 21.12 la représentation de la fonction LISP implantant l'implication.
2. on calcule ensuite la fitness de chacune des fonctions. Dans le cas présent, la fitness est simplement le nombre de fois où la fonction prédit correctement la sortie pour chacune des entrées possibles ;
3. les programmes se reproduisent alors en fonction de leur fitness, comme pour les algorithmes génétiques classiques ;

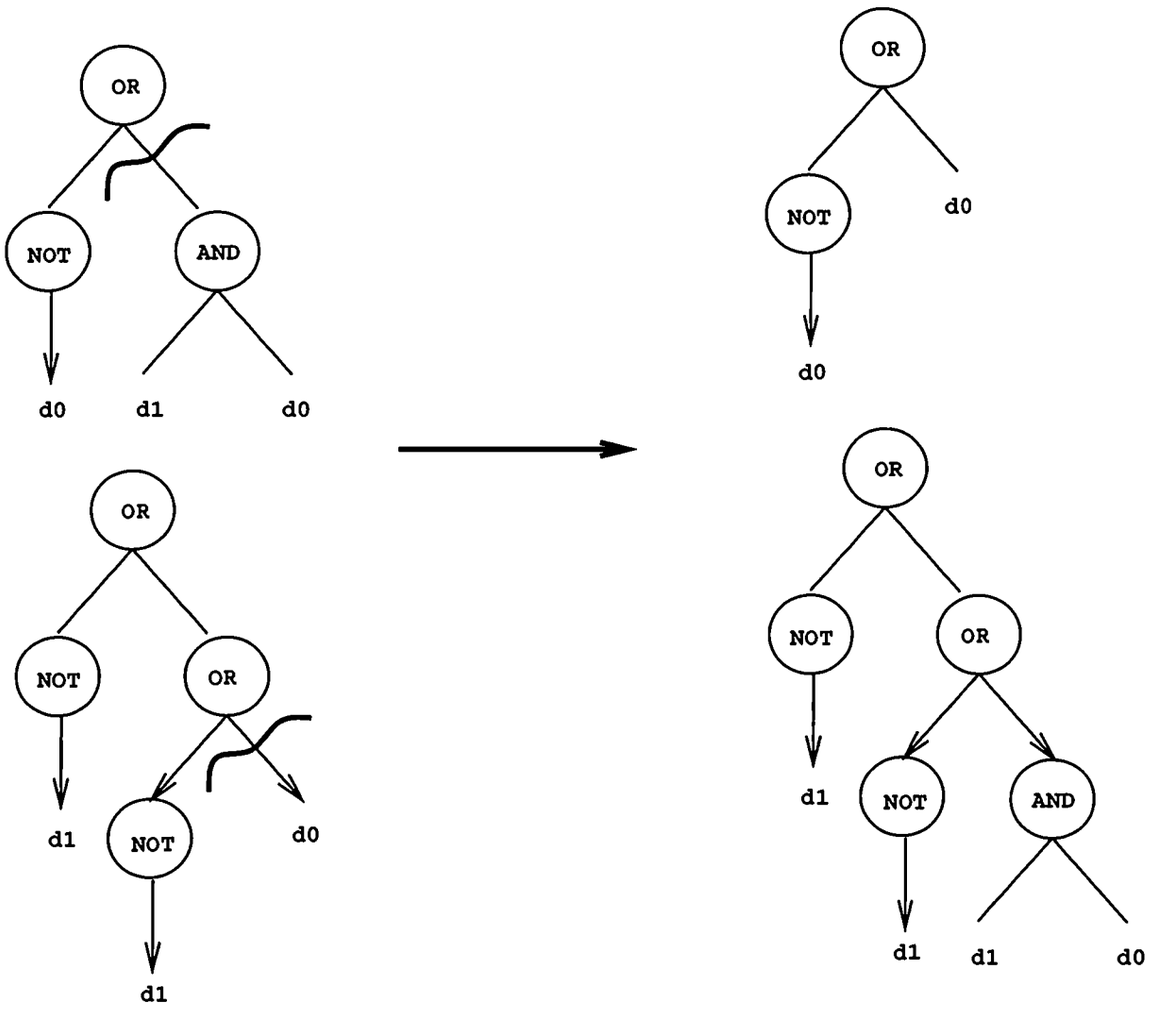


Figure 21.13 – Croisement de deux programmes

4. on effectue ensuite l'opération de croisement. On tire au hasard deux fonctions dans la population et on tire sur chacune de ces deux fonctions un site de croisement au hasard : un site de croisement est une branche, quelconque, de l'arbre représentant la fonction. On remplace alors dans le premier arbre la partie coupée par la partie coupée du second arbre et réciproquement. On peut voir un exemple de croisement sur la figure 21.13 ;

5. on effectue ensuite les mutations. Il s'agit de sélectionner au hasard une fonction, puis un point de mutation dans cette fonction, de couper l'ensemble du sous-arbre situé sous ce point et de le remplacer par un sous-arbre généré aléatoirement ;

6. on peut alors reprendre à l'étape (2) la génération suivante.

Un des problèmes principaux de la programmation génétique est le choix des opérateurs à introduire dans la population initiale. Dans notre cas, le choix de `NOT`, `AND` et `OR` était relativement simple à faire. Dans bien d'autres cas, il n'en est pas de même, et le choix initial des opérateurs conditionne bien souvent la réussite ou l'échec du mécanisme de programmation génétique. Comme dans bien d'autres cas, le savoir-faire de la personne qui met la technique en œuvre conditionne le résultat bien plus sûrement que la technique elle-même.

21.9 Conclusion

Au même titre que les réseaux de neurones, bien que dans une optique différente, les algorithmes génétiques représentent une approche originale et intéressante. Là aussi, il faut se garder de se laisser prendre aux phénomènes de mode et songer à comparer les algorithmes génétiques à des méthodes plus classiques d'optimisation (recuit simulé, branch and bound, etc.) On peut considérer qu'ils sont passés du stade de la recherche pure à celui de la recherche appliquée.

CHAPITRE 22

Apprentissage par renforcement

Frédérick Garcia

22.1 Introduction

L'apprentissage par renforcement peut être défini comme le problème posé à un agent autonome (une personne, un animal, un robot, un système intelligent) évoluant au sein d'un environnement, qui recherche à travers des expériences itérées au sein de cet environnement un comportement optimal.

Ce comportement est modélisé par une stratégie, qui est une fonction associant à l'état courant de l'agent et de l'environnement une action à exécuter. La qualité de ce comportement est évaluée en fonction des récompenses reçues au cours du temps par l'agent suite à l'application de cette stratégie.

L'idée fondamentale de l'apprentissage par renforcement est d'apprendre une bonne stratégie en mettant à jour la stratégie courante après chaque exécution d'une action dans l'environnement, sous la forme d'un renforcement :

$$\text{STRATEGIE}_{n+1} = \text{STRATEGIE}_n + \Delta_n,$$

où Δ_n est une fonction de la récompense reçue.

Pour illustrer cette problématique, considérons l'exemple suivant : Le parking d'un magasin est très grand, et construit autour d'une voie à sens unique qui va de l'entrée du parking jusqu'au magasin. L'objectif est de marcher le moins possible après s'être garé, mais les places sont rares et on veut aussi éviter de devoir aller au parking souterrain du magasin. Optimiser, semaine après semaine, la stratégie de choix d'une place de parking peut être modélisé comme un problème d'apprentissage par renforcement.

Défini de la sorte, en quoi l'apprentissage par renforcement se différentie-t'il des approches déjà existantes en apprentissage ? La réponse réside dans le caractère *local* de cet apprentissage.

Les méthodes de l'apprentissage automatique (voir aussi Chapitre 19) peuvent être cataloguées en fonction du type de *professeur* considéré. Le professeur du système d'apprentissage est le processus qui génèrent les données qui serviront à l'apprentissage. Deux

types majeurs existent en apprentissage automatique, l'apprentissage *supervisé* et l'apprentissage *non-supervisé*. Dans le premier cas, les exemples proposés par le professeur sont du type (x, y) où x est la donnée proprement dite, et y est le label que le professeur associe à x. Le rôle de l'apprentissage est alors de déterminer la règle suivie par le professeur pour associer les labels aux données. Dans le second cas, le professeur ne fournit que des exemples x, et le problème consiste à apprendre des « régularités » dans l'espace des données, afin d'en effectuer une classification. Le professeur ne supervise pas l'apprentissage.

L'apprentissage par renforcement fait intervenir un troisième type de professeur, selon un principe d'essais-erreurs. Intermédiaire entre supervisé et non-supervisé, l'apprentissage par renforcement ne reçoit pas pour chaque donnée x (ici l'état de l'environnement) le label y qui doit lui être associé (ici la meilleure action à exécuter). Toutefois, le professeur fournit pour chaque association proposée (x, y) une note locale r (la récompense) qui indique approximativement la valeur de ce choix. Le rôle du professeur est alors tenu à la fois par l'environnement, à travers la note r qu'il émet, mais aussi par l'agent lui même, qui est donc ici son propre professeur, à travers le choix des couples états-actions qu'il va tester dans cet environnement. Ces deux notions d'essais pour (x, y) et d'erreurs pour r sont véritablement spécifiques à l'apprentissage par renforcement, même si le terme d'apprentissage par essais-erreurs peut parfois improprement être utilisé dans le cadre d'apprentissages supervisés.

Une seconde caractéristique de l'apprentissage par renforcement est relative à son aspect temporel. Généralement, lorsque l'on cherche à classifier des données, l'ordre de présentation des exemples n'influence pas (statistiquement) le résultat. Au contraire, en apprentissage par renforcement, tout choix d'une action exécutée dans un état a des conséquences à plus ou moins long terme, et la donnée de la récompense r d'un couple (x, y) n'est rien sans les autres données qui correspondent à la suite de l'interaction entre l'agent et l'environnement. Par exemple, savoir si c'est une erreur ou non de ne pas se garer maintenant à cet emplacement libre ne peux être décidé sur l'instant. Il faut attendre la suite pour voir si on trouvera effectivement une meilleure place, ou s'il faudra malheureusement visiter le parking souterrain. On a ainsi souvent affaire à des *récompenses retardées*, et les méthodes de l'apprentissage par renforcement offrent des outils permettant de gérer cette difficulté.

Ce caractère local de l'apprentissage par renforcement est fondamental, et distingue clairement cet apprentissage des méthodes évolutionnaires comme la programmation génétiques, les classifieurs, etc., qui peuvent afficher les mêmes objectifs.

On trouvera dans (Sutton and Barto 1998) une bonne présentation de l'historique de l'apprentissage par renforcement, au sein des communautés de l'automatique, de la psychologie et de l'intelligence artificielle.

La plupart des méthodes algorithmiques de l'apprentissage par renforcement sont basées sur des principes simples issus de l'étude de la cognition humaine ou animale, comme par exemple le fait de renforcer la *tendance* à exécuter une action si ses conséquences sont jugées positives, ou encore de faire dépendre ce renforcement de la durée qui sépare la récompense de l'action, ou de la fréquence à laquelle cette action a été testée. Toutefois,

ces techniques algorithmiques ont été formalisées depuis peu dans le cadre des processus décisionnels de Markov, que nous introduisons dans la première partie de ce chapitre. Nous présentons ensuite les principales méthodes de l'apprentissage par renforcement, en s'appuyant sur ce modèle formel. Nous concluons avec quelques exemples d'application.

22.2 Les processus décisionnels de Markov

Les processus décisionnels de Markov (PDM) forment un modèle de la dynamique d'un agent en interaction avec un environnement stochastique à (voir figure 22.1). Dans cette section, nous définissons les notions fondamentales des PDM, et présentons les grandes familles d'algorithmes de résolution.

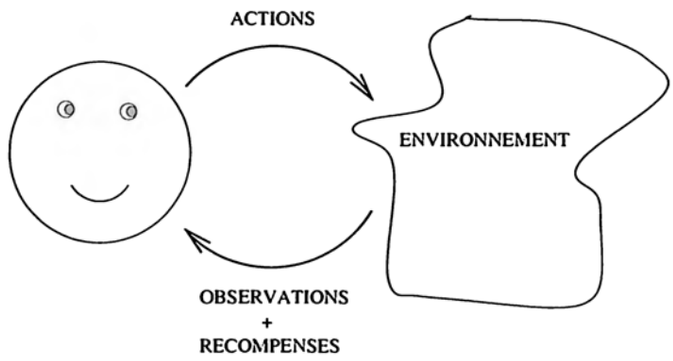


Figure 22.1 – L'agent et l'environnement

22.2.1 Définition d'un processus décisionnel de Markov

Introduisons tout d'abord les concepts principaux intervenant dans la définition des processus décisionnels de Markov.

L'état résume la situation de l'agent et de l'environnement à chaque instant. Il peut se décomposer en :

- un état relationnel agent / environnement,
- un état propre à l'environnement,
- un état interne à l'agent.

Dans l'exemple du parking, un état naturel pour le problème décrit la position de la voiture, et le fait qu'il y ait une place libre ou non à cet endroit.

La dynamique de l'état résulte

- des actions de l'agent sur l'environnement,
- de la dynamique propre de l'environnement,
- de la dynamique interne de l'agent.

Cette dynamique est la plupart du temps non-déterministe, ce qui signifie que les mêmes causes n'entraînent pas toujours les mêmes effets. Toutefois, on suppose que pour chaque action possible la dynamique résultante de cet état est Markovienne, ce qui signifie que la probabilité du nouvel état après exécution de l'action ne dépend que de l'état courant (et de l'action).

Dans le cas du parking, lorsque la voiture avance, sa position change et on peut supposer que la nouvelle place de parking est libre avec une certaine probabilité. Cette dynamique est clairement Markovienne.

Les actions sont choisies et exécutées par l'agent à chaque instant. Suite à cela, il

1. perçoit le nouvel état courant,
2. reçoit une récompense instantanée.

La récompense de l'agent automobiliste pourrait être le scalaire $-i$, où i est la distance entre la place de parking qu'il a choisie et le magasin, ou un coût arbitraire $-C$ s'il a dû se garer au parking souterrain.

Plus formellement, on définit un processus décisionnel de Markov comme un processus stochastique contrôlé satisfaisant la propriété de Markov, assignant des récompenses aux transitions d'états (Puterman 1994). Il est caractérisé par les éléments S, A, T, p et r où

- S est l'espace d'états,
- A est l'espace des actions,
- T est l'axe temporel,
- p_t est une famille de distributions de probabilité sur les transitions d'états,
- r_t est une fonction de récompense sur les transitions d'états.

À chaque instant t de T (voir figure 22.2), l'agent observe l'état courant $s \in S$, applique sur le système une action $a \in A$ qui amène aléatoirement celui-ci dans le nouvel état s', et reçoit une récompense $r \in \mathbb{R}$.

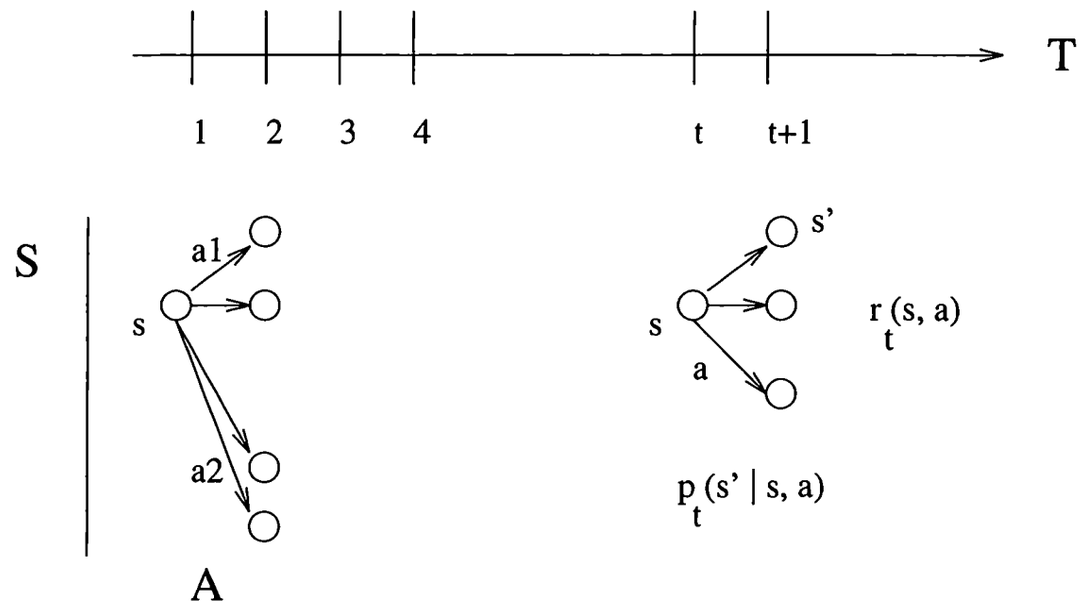


Figure 22.2 – Vue générale d'un PDM

Le domaine T des étapes de décision est dans le cas le plus général un sous-ensemble de la demi-droite $\mathbb{R}^+$. Toutefois, lorsque T est continu, les problèmes de décisions séquentielles correspondants sont mieux traités par les méthodes de la théorie du contrôle optimal, sur la base des équations de la dynamique du système ; nous considérerons ici uniquement le cas discret. Pour T discret, l'ensemble des étapes de décision peut être fini ou infini (on parle d'horizon fini ou d'horizon infini), et ces étapes peuvent correspondre à des instants prédéterminés sur $\mathbb{R}^+$, aléatoires ou dépendants de certains états.

Les domaines S et A sont supposés finis, même si de nombreux résultats peuvent être étendus aux cas S et A continus ou énumérables. L'ensemble A est en fait souvent

défini comme la réunion d'ensembles A_s, spécifiant pour chaque état $s \in S$ les actions exécutables en ce point. En pratique, il est équivalent et plus simple de ne considérer formellement qu'un unique ensemble A.

Les distributions p_t, nommées probabilités de transition, caractérisent la dynamique de l'état du système. Pour une action a fixée, $p_t(s'|s,a)$ représente la probabilité que le système passe dans l'état s' après avoir exécuté à la date t l'action a dans l'état s. On a $\forall s, a, \sum_{s'} p(s'|s,a) = 1$.

Les distributions p_t vérifient la propriété fondamentale qui donne son nom aux processus décisionnels de Markov considérés ici. Si on note h_t l'historique à la date t du processus (s_t, a_t), $h_t = (s_0, a_0, \ldots, s_{t-1}, a_{t-1}, s_t)$, alors $\forall h_t, s_{t+1}$, $P(s_{t+1} \mid h_t, a_t) = P(s_{t+1} \mid s_t, a_t) = p_t(s_{t+1} \mid s_t, a_t)$. Il faut noter que cela n'implique pas nécessairement que le processus stochastique induit (s_t) soit lui même Markovien, tout dépend de la stratégie de choix des actions a_t.

Comme résultat d'avoir choisi l'action a dans l'état s, l'agent décideur reçoit une récompense, ou revenu, $r_t(s,a) \in \mathbb{R}$. Les valeurs de r_t positives peuvent être considérées comme des gains, et les valeurs négatives comme des coûts. Cette récompense peut être instantanément perçue à la date t, ou accumulée de la date t à la date $t+1$, l'important est qu'elle ne dépende que de l'état et de l'action choisie à l'instant courant.

Deux extensions sont classiquement rencontrées. Tout d'abord, $r_t(s,a)$ peut être aléatoire. On considère alors souvent sa valeur moyenne $\bar{r}_t(s,a)$. D'autre part, r_t peut aussi dépendre de l'état d'arrivée s'. La récompense $r_t(s,a,s')$ est alors aléatoire à la date t, et sa valeur moyenne est $\bar{r}_t(s,a) = \sum_{s'} p_t(s'|s,a) r_t(s,a,s')$. Dans tous les cas, on suppose r_t bornée dans $\mathbb{R}$ (ainsi, $r_t = t$ n'est pas acceptable en horizon infini).

Enfin, un cas particulier de PDM couramment rencontré est celui des processus *stationnaires*, où les probabilités de transitions p_t et les récompenses r_t ne dépendent pas du temps :

$$\forall t \in T \quad p_t() = p(),\ r_t() = r().$$

22.2.2 Les stratégies et les critères de performance

Une stratégie est la procédure qui permet à l'agent, étant données les observations passées et présentes, de choisir à chaque instant t l'action à exécuter dans l'environnement afin de tenter d'en contrôler la dynamique. Deux distinctions sont essentielles ici. Tout d'abord, une stratégie peut déterminer précisément l'action à effectuer, ou simplement définir une distribution de probabilité selon laquelle cette action doit être sélectionnée. Ensuite, une stratégie peut se baser sur l'historique h_t du processus, ou peut ne simplement considérer que l'état courant s_t. Nous obtenons ainsi 4 familles distinctes de stratégies, comme indiqué sur le tableau 22.1 :

Processus	Stratégie π_t	déterministe	aléatoire
PDM	Markovienne	$s_t \longrightarrow a_t$	$s_t \longrightarrow u_t \in \mathcal{P}(A)$
	histoire-dépendante	$h_t \longrightarrow a_t$	$h_t \longrightarrow u_t \in \mathcal{P}(A)$

Table 22.1 – Différentes familles de stratégie pour les PDM.

Indépendamment de cela, et comme pour le processus lui-même, la définition des stratégies peut ou non dépendre explicitement du temps. Ainsi, une stratégie est *stationnaire* si $\forall t\ \pi_t = \pi$. Le modèle le plus simple de stratégie décisionnelle, et pourtant le plus rencontré dans les applications, correspond alors aux stratégies *stationnaires déterministes Markoviennes* :

Définition 22.1 – stratégies stationnaires déterministes Markoviennes –

$$\pi : S \longrightarrow A.$$

Se poser un problème décisionnel de Markov, c'est rechercher parmi une famille de stratégies celles qui optimisent un critère de performance donné pour le processus décisionnel Markovien considéré. Au sein des PDM, ce critère a toujours pour ambition de caractériser les stratégies qui permettront lors de leur exécution de générer des séquences de récompenses les plus importantes possibles. En termes formels, cela revient toujours à évaluer une stratégie sur la base d'une mesure du cumul espéré des récompense instantanées le long d'une trajectoire, comme on peut le voir pour le critère γ-pondéré,

$$E[\sum_{t=0}^{\infty} \gamma^t r_t] \text{ (critère } \gamma\text{-pondéré)},$$

et pour le critère fini :

$$E[\sum_{t=0}^{N-1} r_t] \text{ (critère fini)},$$

les plus étudiés au sein de la théorie des PDM.

Dans ces définitions, $E[.]$ dénote l'espérance mathématique sur l'ensemble des réalisations du processus décisionnel Markovien. Une caractéristique commune à ces critères est leur aspect additif en r_t, qui est un cas particulier d'une propriété plus générale dite de *temps séparabilité*, vérifiée aussi par exemple pour une forme produit des r_t si ces derniers sont positifs ou nuls. Cette propriété est une condition nécessaire au principe d'optimalité, dit de *cohérence dynamique* (Bacharach and Hurley 1991), à la base des nombreux algorithmes de programmation dynamique permettant de résoudre efficacement les PDM.

22.2.3 Optimalité et fonctions de valeur

Afin de comparer différentes stratégies selon ces critères, on définit pour chacun d'entre eux une *fonction de valeur* qui, pour une stratégie π fixée, associe à tout état initial $s \in S$ la valeur espérée du critère considéré en suivant π à partir de s

$$\forall \pi \quad V^\pi : S \longrightarrow \mathbb{R}.$$

Étant donnée une fonction de valeur V^π déduite du critère retenu, la préférence d'une stratégie sur une autre est alors définie par :

Définition 22.2 – Préférence entre fonctions de valeur – *La stratégie π_1 domine la stratégie π_2 si*

$$\forall s_0 \quad V^{\pi_1}(s_0) \geq V^{\pi_2}(s_0).$$

Ainsi, la recherche d'une stratégie optimale se transforme-t'elle en un problème d'optimisation de la fonction de valeur V^π : on recherche une stratégie optimale

$$\pi^* = \underset{\pi}{\operatorname{argmax}}\, V^\pi.$$

Les deux critères énoncés conduisent aux fonctions de valeurs suivantes :

Définition 22.3 – Le critère fini – *Si* $T = \{0, \ldots, N-1\}$ *on pose*

$$\forall s \in S \quad V_N^\pi(s) = E_\pi[\sum_{t=0}^{N-1} r_t \mid s_0 = s].$$

Dans cette définition, $E_\pi[.]$ dénote l'espérance mathématique sur l'ensemble des réalisations du processus en suivant la stratégie π.

Il est classique pour ce problème à horizon fini d'associer une valeur terminale $h(s_N)$ au dernier état rencontré, qui se rajoute alors au critère.

Définition 22.4 – Le critère γ-pondéré – *La fonction de valeur d'une stratégie* π *en chaque état* s *est définie par*

$$\forall s \in S \quad V_\gamma^\pi(s) = E_\pi[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s],$$

où $0 \leq \gamma < 1$.

Ce terme γ, *discount factor* en anglais, représente la valeur à la date t d'une unité de récompense reçue à la date $t+1$. Reçue à la date $t+\tau$, cette même unité vaudrait γ^τ.

Le facteur γ a pour principal intérêt d'assurer la convergence de la série en horizon infini. D'un point de vue pratique, il est naturellement utilisé au sein des MDP modélisant des processus de décision économique.

Il est toutefois possible de choisir $\gamma = 1$ lorsque la maximisation du critère garde un sens. C'est en particulier le cas si pour toute stratégie et tout état initial, le processus Markovien (s_t) conduit sûrement en un état absorbant $s_{abs} \in S$ (*i.e.*, $p(s_{abs} \mid s_{abs}, \pi(s_{abs})) = 1$) de gain nul (*i.e.*, $r(s_{abs}, \pi(s_{abs})) = 0$).

On considère alors

$$V^\pi(s) = E_\pi[\sum_{t=0}^{\infty} r_t \mid s_0 = s] = \lim_{N \to \infty} V_N^\pi(s)$$

que l'on nomme « critère total » et qui correspond à un critère fini à horizon aléatoire.

L'exemple du parking illustre bien ce cas. L'état absorbant correspond à l'état où la voiture est garée, c'est la fin du problème.

Équations d'optimalité pour le critère γ-pondéré

Nous considérons ici le cas du critère γ-pondéré pour des processus décisionnels de Markov stationnaires, qui est le plus couramment rencontré dans les applications de l'apprentissage par renforcement.

D'une manière générale, la recherche de $\pi^* = \text{argmax}_\pi V^\pi$ est étroitement liée au calcul de $V^* = \max_\pi V^\pi$. En effet, un résultat fondamental des PDM est l'existence d'une équation d'optimalité caractérisant entièrement cette fonction de valeur optimale, que l'on nomme aussi *équation de Bellman* (Bellman 1957). Dans le cadre du critère γ-pondéré, cette équation prend la forme :

$$\forall s \in S \quad V^*(s) = \max_{a \in A} \{r(s,a) + \gamma \sum_{s' \in S} p(s' \mid s,a) V^*(s')\}. \tag{22.1}$$

On montre alors que la solution V^* de cette équation est unique, et surtout que la connaissance de cette fonction de valeur optimale V^* permet d'obtenir une stratégie optimale π^*, car on a

$$\forall s \in S \quad \pi^*(s) = \text{argmax}_{a \in A} \{r(s,a) + \gamma \sum_{s' \in S} p(s' \mid s,a) V^*(s')\} \tag{22.2}$$

avec

$$V^* = V^{\pi^*}.$$

On constate ainsi que les stratégies optimales pour le critère γ-pondéré appliqué à des processus décisionnels de Markov stationnaires sont des stratégies *stationnaires déterministes Markoviennes.*

Cas du critère fini

Pour le critère fini, il apparaît que l'aspect stationnaire des stratégies et de la fonction de valeur optimale n'est plus vérifié. La meilleure action à exécuter à chaque instant dépend de l'état courant, mais aussi du nombre d'étapes restant à parcourir. Pour cela, on définit à chaque instant $i = 0, \ldots, N-1$ une stratégie π_i et une fonction de valeur associée $V_i = V^{\pi_i}$. On a alors N équations d'optimalité

$$\begin{aligned} \forall s \in S \quad V_i^*(s) &= \max_{a \in A} \{r(s,a) + \sum_{s' \in S} p(s' \mid s,a) V_{i+1}^*(s')\} \quad \forall i = 0, \ldots, N-1, \\ \text{avec } V_N^*(s) &= h(s). \end{aligned}$$

22.2.4 Algorithmes de résolution des PDM

Commençons par le critère à horizon fini. Ce cas est très simple, et l'algorithme classique consiste en une méthode de type itératif sur la fonction de valeur, en partant de la dernière étape. Si L et M sont les tailles respectives de S et A, et N l'horizon, la complexité de l'algorithme est en $O(NML^2)$.

Une très importante littérature existe sur la résolution des PDM pour le critère γ-pondéré. Nous n'évoquons ici que les grandes classes de méthodes, et invitons le lecteur à se référer aux ouvrages de référence sur le domaine (Puterman 1994; Bertsekas and Tsitsiklis 1996). Notons juste que les PDM correspondent à des problèmes P-complets pour les critères classiques (Papadimitriou and Tsitsiklis 1987)

Une des premières approches est basée sur la programmation linéaire (D'Epenoux 1963). Le principe consiste à minimiser $\sum_{s\in S} V(s)$ sous les contraintes

$$V(s) \geq r(s,a) + \gamma \sum_{s'\in S} p(s' \mid s,a)V(s'), \quad \forall s \in S,\ a \in A.$$

Si $p()$ et $r()$ sont codées sur B bits, la complexité d'un tel algorithme de programmation linéaire sur les rationnels est polynomiale en L, M, B, avec des temps de résolution extrêmement lents.

L'approche la plus classique se base aussi sur la résolution directe de l'équation d'optimalité de Bellman, en utilisant pour cela une méthode itérative de type point fixe, d'où son nom anglais de *value iteration* (Bellman 1957; Bertsekas 1987; Puterman 1994).

La solution de l'équation 22.1 est obtenue comme limite de la suite :

$$\forall s \in S \quad V_{n+1}(s) = \max_{a\in A}\{r(s,a) + \gamma \sum_{s'\in S} p(s' \mid s,a)V_n(s')\} \tag{22.3}$$

avec $V_0()$ initialisée à 0 par exemple. Il est établi que cet algorithme itératif converge en un nombre maximum d'itérations polynomial en L, M, B et $1/(1-\gamma)\log(1/(1-\gamma))$, chaque itération étant de complexité $O(ML^2)$ (Littman *et al.* 1995).

L'autre classe importante d'algorithmes de résolution est constituée des méthodes couplant itération sur la fonction de valeur et itération sur la stratégie elle-même, dites de *policy iteration.*

Considérons une stratégie π. On peut montrer que sa fonction de valeur V^π est la solution du système d'équations linéaires

$$\forall s \in S \quad V^\pi(s) = r(s,\pi(s)) + \gamma \sum_{s'\in S} p(s' \mid s,\pi(s))V^\pi(s'). \tag{22.4}$$

Soit alors π' la stratégie obtenue à partir de la fonction de valeur V^π de π de la manière suivante (*roll-out policy*) :

$$\forall s \in S \quad \pi'(s) = \operatorname*{argmax}_{a\in A}\{r(s,a) + \gamma \sum_{s'\in S} p(s' \mid s,a)V^\pi(s')\}.$$

On montre que la stratégie π' domine la stratégie π, c'est à dire que $\forall s \quad V^{\pi'}(s) \geq V^\pi(s)$, l'égalité n'étant obtenue que si $V^\pi = V^{\pi'} = V^*$.

L'algorithme d'itération de la stratégie se décline donc ainsi : soit la stratégie π_n à l'itération n. Dans une première étape on résout le système d'équations linéaires

$$\forall s \in S \quad V_n(s) = r(s,\pi_n(s)) + \gamma \sum_{s'\in S} p(s' \mid s,\pi_n(s))V_n(s')$$

puis dans un second temps on améliore la stratégie courante en posant

$$\forall s \in S \quad \pi_{n+1}(s) = \underset{a \in A}{\operatorname{argmax}} \{ r(s,a) + \gamma \sum_{s' \in S} p(s' \mid s,a) V_n(s') \}. \tag{22.5}$$

La complexité de l'algorithme d'itération de la stratégie est en $O(ML^2) + O(L^3)$ par itération, avec un nombre maximum d'itérations polynomial en L, M, B à γ constant. Expérimentalement, la méthode d'itération de la stratégie apparaît plus efficace que l'itération de la valeur (Puterman 1994).

Les deux derniers algorithmes sont chacun à la base d'une famille de variantes reprenant le même principe d'itération sur la fonction de valeur ou sur la stratégie. Citons ainsi les algorithmes *modified policy iteration* (Puterman and Shin 1978), *asynchronous dynamic programming* (Bertsekas 1987) ou *real-time dynamic programming* (Barto *et al.* 1995), qui tous amènent des améliorations. Nous allons voir que l'apprentissage par renforcement doit être aussi vu comme un développement tendant à améliorer ces méthodes de base.

22.2.5 Observabilité du processus

Il est possible de définir une modélisation plus générale des processus décisionnels de Markov en tenant compte des propriétés d'observabilité de l'état s_t. Pour cela, on suppose qu'en chaque état l'agent observe o_t parmi un ensemble O, et qu'il existe une relation liant l'observation courante à l'état courant et à l'action qui vient d'être exécutée, que l'on représente à travers la probabilité conditionnelle $P(o_t \mid s_t, a_{t-1}) = q_t(o_t \mid s_t, a_{t-1})$. L'ensemble O et les fonction q_t sont un modèle du système d'information de l'agent. On distingue alors :

- les processus décisionnels de Markov *complètement observables* (PDM) : $O = S$ et $P(o_t \mid s_t, a_{t-1}) = 1$ ssi $o_t = s_t$
- les processus décisionnels de Markov *non observables* (PDMNO) : $O = \{\}$, i.e. $P(s_t \mid o_t) = P(s_t)$
- les processus décisionnels de Markov *partiellement observables* (PDMPO)

Dans le cas général d'un PDMPO, plusieurs types de stratégies optimales peuvent être recherchés (Lovejoy 1991), étendant ce que nous avons vu pour les PDM. Ici, l'observation o_t remplace l'état s_t, et ho_t représente l'historique des observations et actions passées : $ho_t = (s_0, o_0, \ldots, o_{t-1}, a_{t-1}, o_t)$. Le choix d'une action à la date t peut ainsi se baser sur o_t, sur ho_t, sur une approximation $\hat{s}_t$ de s_t (méthodes de type reconstruction d'état), ou sur un état de croyance de l'agent modélisé par une distribution de probabilité $p_t \in \mathcal{P}(S)$.

Un récapitulatif des différentes stratégies décisionnelles pour les PDMPO est donné table 22.2.

Ajoutons que d'un point de vue théorique, il est possible de montrer que pour de nombreux critères d'optimalité comme ceux énoncés en section 22.2.2, les stratégies optimales sont de type croyance-dépendante déterministes. Il est alors nécessaire d'un point de vue pratique de construire une *représentation finie* de la fonction de valeur sur les états de croyance. Que ce soit par approche exacte ou approchée, la résolution reste très complexe et seuls des problèmes de très faible taille peuvent être résolus (Kaelbling *et al.* 1998).

Processus	Stratégie π_t	déterministe	aléatoire
PDM	Markovienne	$s_t \longrightarrow a_t$	$s_t \longrightarrow u_t \in \mathcal{P}(A)$
	histoire-dépendante	$h_t \longrightarrow a_t$	$h_t \longrightarrow u_t \in \mathcal{P}(A)$
PDMPO	pseudo-Markovienne	$o_t \longrightarrow a_t$	$o_t \longrightarrow u_t \in \mathcal{P}(A)$
	pseudo-Markovienne	$\hat{s}_t \longrightarrow a_t$	$\hat{s}_t \longrightarrow u_t \in \mathcal{P}(A)$
	histoire-dépendante	$ho_t \longrightarrow a_t$	$ho_t \longrightarrow u_t \in \mathcal{P}(A)$
	croyance-dépendante	$p_t \in \mathcal{P}(S) \longrightarrow a_t$	$p_t \in \mathcal{P}(S) \longrightarrow u_t \in \mathcal{P}(A)$
PDMNO	plan a priori	$(a_0, a_1, \ldots, a_t)$	$(u_0, u_1, \ldots, u_t)$

Table 22.2 – Récapitulatifs des différentes familles de stratégies.

Enfin, dans le cas où l'on recherche des stratégies pseudo-Markoviennes basées sur la simple observation courante o_t, donc sans mémoire du passé, on montre que l'on gagne à rechercher des stratégies aléatoires (Singh *et al.* 1994).

22.3 Les méthodes de l'apprentissage par renforcement

Comme nous l'avons vu dans la section introductive de ce chapitre, l'apprentissage par renforcement consiste à apprendre un comportement optimal à travers une séquence d'expériences au sein d'un environnement, ces expériences consistant à agir dans un état donné, et à observer le nouvel état résultant et la récompense ou punition instantanée associée. Aujourd'hui, l'apprentissage par renforcement peut être considéré comme une approche permettant de dépasser les techniques classiques de résolution des processus décisionnels de Markov selon deux directions :

- L'emploi de simulations de la dynamique du processus à contrôler, afin d'orienter l'exploration de l'espace des fonctions de valeurs ou des stratégies. Cela est traduit en pratique par l'emploi d'algorithmes itératifs stochastiques caractéristiques de l'apprentissage par renforcement.
- L'emploi de représentations structurées et compactes des fonctions de valeur et des stratégies, permettant d'aborder ainsi la résolution de problèmes décisionnels de très grande taille impossible à traiter en programmation dynamique classique.

Historiquement toutefois, le premier axe a prévalu, et les méthodes que nous allons présenter dans cette section relèvent de cet aspect. Nous réservons la section suivante 22.4 aux approches traitant du problème des représentations compactes en apprentissage par renforcement[1].

---

1 Le terme *neuro-dynamic programming*, que nous pourrions traduire par *programmation neuro-dynamique*, ou *neuro-progammation dynamique*, a été introduit par Bertsekas et Tsitsiklis (Bertsekas and Tsitsiklis 1996) pour définir l'ensemble des techniques couplant programmation dynamique ou apprentissage par renforcement et emploi de représentations compactes. Nous continuerons toutefois de parler d'apprentissage par renforcement, à partir du moment où la simulation du processus à contrôler est un élément clé des algorithmes de résolution, qu'un modèle du système soit utilisé ou non.

22.3.1 Méthodes directes et indirectes

Selon la présence ou non du maintien d'un modèle explicite de la dynamique du processus markovien que l'on cherche à contrôler (déterminé par les probabilités de transition $p(s' \mid s, a)$ et les revenus $r(s, a)$), les méthodes d'apprentissage par renforcement sont dites directes ou indirectes. À ce titre, les méthodes de programmation dynamique de la section précédente doivent être considérées comme des méthodes directes, où le modèle maintenu est le modèle exact par hypothèse.

Lorsque le modèle de la dynamique n'est pas connu initialement, les méthodes indirectes doivent donc estimer en ligne le modèle dynamique. Dans les cas discrets que nous considérons, cela est fait simplement en estimant $p(s' \mid s, a)$ par $\frac{n_{sa}^{s'}}{n_{sa}}$, soit le rapport entre le nombre de fois où l'exécution de a dans s a conduit en s' sur le nombre de fois où l'action a a été choisie dans s. Dans l'exemple du parking il s'agirait simplement d'estimer les probabilités d'occupation des places. D'autre part, il s'agit de rechercher sur la base du modèle courant une fonction de valeur ou une stratégie optimale. Il est possible alors d'exploiter les algorithmes classiques de programmation dynamique. La question principale de ce type d'approche est donc liée tout naturellement au couplage entre estimation, optimisation et simulation (ou encore exécution) : faut il attendre un modèle le plus exact possible avant d'entamer la phase d'optimisation ? peut on entrelacer au maximum estimation et optimisation, c'est à dire modifier à chaque transition observée la fonction de valeur et le modèle estimée ? Il semble actuellement que cette dernière approche soit préférable, comme par exemple dans RTDP (*real-time dynamic programming* (Barto *et al.* 1995), où estimation, programmation dynamique et exécution sont concurrents. Parmi les algorithmes de ce type les plus connus en apprentissage par renforcement, citons également *Dyna* (Sutton 1990), *Queue-Dyna* (Peng and Williams 1993) et *Prioritized Sweeping* (Moore and Atkeson 1993).

Les méthodes directes ne passent pas par l'estimation d'un modèle de la dynamique du système : les paramètres cachés $p(s' \mid s, a)$ et $r(s, a)$ ne sont pas estimés, et seule la fonction de valeur est mise à jour itérativement au cours du temps. Le principal avantage est ici en terme de place mémoire nécessaire, et historiquement, l'apprentissage par renforcement revendiquait d'être une méthode directe. Ainsi, les deux algorithmes les plus classiques de l'apprentissage par renforcement, le Q-learning et TD(λ), que nous présentons ci-dessous partagent, chacun à sa manière, le même principe général qui est d'approximer à partir d'expériences (état_n, action_n, état_n, récompense_n) une fonction de valeur optimale V

$$V_{n+1} = V_n + \Delta(s_n, a_n, s_{n+1}, r_n)$$

sans nécessiter la connaissance a priori d'un modèle du processus, et sans chercher à estimer ce modèle à travers les expériences accumulées. Toutefois, il semble que l'absence d'un modèle explicite lors de l'apprentissage ne soit plus une bonne caractérisation des algorithmes d'apprentissage par renforcement, et qu'il faille aussi y inclure maintenant des algorithmes comme RTDP, ou des algorithmes de programmation dynamique basés sur une représentation paramétrée de la fonction de valeur, même si la dynamique du système est parfaitement connue initialement.

22.3.2 L'algorithme Q-learning

Dans le cas où le domaine n'est pas connu initialement, l'algorithme Q-learning est une méthode d'apprentissage par renforcement permettant de résoudre l'équation de Bellman 22.1 pour le critère γ-pondéré. Il est le plus utilisé en pratique, du fait de sa simplicité. Son principe consiste à mettre à jour itérativement les valeurs de la fonction V^* recherchée, sur la base de l'observation des transitions instantanées et des revenus associés.

La forme de l'équation 22.1 n'est pas satisfaisante pour en dériver directement un algorithme adaptatif de résolution. Pour cela, Watkins (Watkins 1989) a introduit la fonction de valeur Q, dont la donnée est équivalente à celle de V.

Définition 22.5 – Fonction de valeur Q – *À une stratégie π fixée de fonction de valeur V^π, on associe la nouvelle fonction*

$$\forall s \in S, a \in A \quad Q^\pi(s,a) = r(s,a) + \gamma \sum_{s'} p(s' \mid s,a) V^\pi(s').$$

L'interprétation de la valeur $Q^\pi(s,a)$ est la suivante : c'est la valeur espérée du critère pour le processus partant de s, exécutant l'action a, puis suivant la stratégie π par la suite. Il est clair que $V^\pi(x) = Q^\pi(x, \pi(x))$, et l'équation de Bellman vérifiée par la fonction Q^* devient :

$$\forall s \in S, a \in A \quad Q^*(s,a) = r(s,a) + \gamma \sum_{s'} p(s' \mid s,a) \max_b Q^*(s',b).$$

On a alors

$$\begin{aligned} \forall s \in S \quad V^*(s) &= \max_a Q^*(s,a), \\ \pi^*(s) &= \operatorname*{argmax}_a Q^*(s,a). \end{aligned}$$

Le principe de l'algorithme Q-learning est de mettre à jour à la suite de chaque transition (s_n, a_n, s_{n+1}, r_n) la fonction de valeur courante Q_n pour le couple (s_n, a_n), où s_n représente l'état courant, a_n l'action sélectionnée et réalisée, s'_n l'état résultant et r_n la récompense immédiate.

Dans cet algorithme, N_{tot} est un paramètre initial fixant le nombre d'itérations. Le taux d'apprentissage $\alpha_n(s,a)$ est propre à chaque paire état-action, et décroît vers 0 à chaque passage. La fonction Simuler retourne le nouvel état et la récompense associée selon la dynamique du système. Le choix de l'état courant et de l'action à exécuter est effectué par les fonctions ChoixEtat et ChoixAction et sera discuté plus loin. La fonction Initialiser revient la plupart du temps à initialiser les composantes de Q_0 à 0.

Il est immédiat d'observer que l'algorithme Q-learning est une formulation stochastique de l'algorithme de *value iteration* pour les PDM (équation 22.3). Ce dernier peut en effet s'exprimer directement en terme de Q-valeurs :

Algorithme 16: Le Q-learning

Initialiser(Q_0);
pour $n \leftarrow 0$ **Jusqu'à** N_{tot} **faire**
  $s_n \leftarrow$ ChoixEtat ;
  $a_n \leftarrow$ ChoixAction ;
  $(s'_n, r_n) \leftarrow$ Simuler(s_n, a_n);
  { mise à jour de Q_n };
  début
    $Q_{n+1} \leftarrow Q_n$;
    $d_n \leftarrow r_n + \gamma \max_b Q_n(s'_n, b) - Q_n(s_n, a_n)$;
    $Q_{n+1}(s_n, a_n) \leftarrow Q_n(s_n, a_n) + \alpha_n(s_n, a_n) d_n$;
  fin
retourner Q_n;

$$
\begin{aligned}
V_n(s) &= \max_{a \in A} Q_n(s,a), \\
V_{n+1}(s) &= \max_{a \in A} \overbrace{\left\{ r(s,a) + \gamma \sum_{s' \in S} p(s' \mid s,a) V_n(s') \right\}}^{Q_n(s,a)} \\
\Rightarrow \quad Q_{n+1}(s,a) &= r(s,a) + \gamma \sum_{s' \in S} p(s' \mid s,a) \max_{a' \in A} Q_n(s',a').
\end{aligned}
$$

Le Q-learning est alors obtenu en remplaçant le terme $r(s,a) + \sum_{s'} p(s' \mid s,a) \max_{a' \in A} Q_n(s',a')$ par son estimation la plus simple construite à partir de la transition courante $r_n + \max_{a'} Q_n(s'_n, a')$.

La convergence de cet algorithme a été bien étudiée et est maintenant établie (Watkins 1989; Jaakkola *et al.* 1994). Sous les hypothèses de finitude de S et A, si l'on suppose que chaque paire (s,a) est visitée un nombre infini de fois, si $\sum_n \alpha_n(s,a) = \infty$ et $\sum_n \alpha_n^2(s,a) < \infty$, et enfin si $\gamma < 1$ ou si $\gamma = 1$ et pour toute stratégie il existe un état absorbant de revenu nul, alors la fonction Q_n converge presque sûrement vers Q^*. Rappelons que cette convergence presque sûre signifie que $\forall s, a$ la suite $Q_n(s,a)$ converge vers $Q^*(s,a)$ avec une probabilité égale à 1. En pratique, la suite $\alpha_n(s,a)$ est souvent définie comme

$$\alpha_n(s,a) = \frac{1}{n_{sa}}.$$

22.3.3 Le dilemme exploration/exploitation

Nous avons vu que la convergence vers l'optimum de l'algorithme Q-learning était assurée sous la seule contrainte de parcourir indéfiniment l'ensemble des états et actions possibles. En pratique toutefois, des heuristiques sur le choix de s_n et a_n sont pris en compte.

La plupart du temps, le choix le plus naturel concernant s_n est de poser à chaque itération $s_{n+1} = s'_n$, c'est à dire de laisser à la dynamique du système le soin de gérer l'exploration de l'espace d'états. Cela permet d'une part de se concentrer sur les zones importantes de l'espace d'états, accélérant ainsi la convergence (c'est aussi une des raisons de l'efficacité d'algorithmes comme RTDP). C'est d'autre part souvent nécessaire du fait de la structure des simulateurs utilisés lors de l'apprentissage. Il est clair toutefois que lorsque s'_n est un état absorbant, il est nécessaire de réinitialiser le processus en tirant par exemple au hasard un nouvel état s_{n+1} dans S.

La seconde heuristique concerne le choix de l'action a_n. Une action choisie uniformément dans A à chaque itération satisfait bien le critère de convergence, avec une *exploration* maximale, mais est peu efficace pour deux raisons. Tout d'abord, la valeur de la meilleure action en chaque état étant aussi souvent mis à jour que la valeur de la plus mauvaise action, l'apprentissage se fait sûrement mais très lentement. D'autre part, les revenus accumulés au cours de l'apprentissage sont nécessairement assez moyens, ce qui peut être inacceptable lorsque cet apprentissage est en prise au système dynamique réel et non simulé.

Inversement, le choix à chaque itération de l'action a_n correspondant à la stratégie optimale courante $\pi_{Q_n}(s) = \text{argmax}_a Q_n(s,a)$ (on parle de *greedy-action* et de *greedy-policy*, pour une méthode dite *optimiste* selon un principe d'*exploitation*) n'est pas non plus satisfaisant, car il conduit généralement soit à une stratégie sous-optimale, soit à la divergence de l'algorithme. Ainsi les implémentations du Q-learning retiennent un compromis entre exploration et exploitation, qui consiste à suivre la stratégie optimale courante π_{Q_n} la plupart du temps, tout en choisissant plus ou moins régulièrement une action aléatoire pour a_n. Plusieurs méthodes de choix de a_n ont été proposées, que l'on classe en deux catégories dites méthodes dirigées ou non-dirigées (Thrun 1992).

Les méthodes non-dirigées utilisent peu d'information issu de l'apprentissage autre que la fonction Q elle même. Citons par exemple (Bertsekas and Tsitsiklis 1996; Kaelbling *et al.* 1996) :

- sur un intervalle de N_1 itérations, suivre $\pi_{Q_n}(s_n)$, puis sur N_2 itérations, tirer uniformément a_n dans A ;
- utiliser à chaque itération un tirage semi-uniforme, qui consiste à suivre $\pi_{Q_n}(s_n)$ avec une probabilité $1-\tau$, ou à tirer uniformément a_n dans A avec une probabilité τ, et $\tau \in [0,1]$;
- tirer a_n dans A selon une distribution de Boltzmann, la probabilité associée à l'action a étant

$$p_T(a) = \frac{\exp(-\frac{Q_n(s_n,a)}{T})}{\sum_{a'} \exp(-\frac{Q_n(s_n,a')}{T})} \quad .$$
$$\text{avec } \lim_{n\to\infty} T = 0$$

Ces différentes fonctions d'exploration font intervenir des paramètres (N_1, N_2, τ et T) qui contrôlent le degré d'exploration dans le choix de a_n. Par exemple, si $N_1 = 0$, $T \to +\infty$ ou $\tau = 1$, l'algorithme d'apprentissage n'oriente pas sa direction d'apprentissage dans l'espace des stratégies, qu'il parcourt uniformément. Les cas utiles en pratique sont toutefois pour $N_1 > 0$, $T < +\infty$ et $\tau < 1$, qui assurent expérimentalement une convergence beaucoup plus rapide.

Les méthodes dirigées utilisent pour leur part des heuristiques propres au problème de l'exploration, en se basant sur des informations acquises au cours de l'apprentissage. La plupart de ces méthodes reviennent à ajouter à la valeur $Q(s,a)$ d'une action dans

un état un bonus d'exploration (Meuleau 1996). Ce bonus peut être local comme dans la méthode de l'*interval estimation* (Kaelbling 1993), où propagé d'état à état au cours de l'apprentissage (Meuleau and Bourgine 1998). Des définitions simples de ce bonus d'exploration conduisent toutefois à des résultats déjà intéressants :

- la *recency-based* méthode : le bonus est égal à $\epsilon\sqrt{\delta n_{sa}}$ où δn_{sa} représente le nombre d'itérations parcouru depuis la dernière exécution de l'action a dans l'état s, et où ϵ est une constante inférieure à 1 ;
- la méthode de l'*uncertainty estimation* : le bonus est égal à $\frac{c}{n_{sa}}$, où c est une constante et n_{sa} représente le nombre de fois où l'action a a déjà été choisie dans l'état s.

Ces méthodes reviennent à définir une fonction de valeur de l'exploration. Ces Q-fonctions de valeur peuvent en fait être directement apprises par un algorithme de type Q-learning, comme le proposent Wiering et Schmidhuber (Wiering and Schmidhuber 1998).

Constatons pour finir qu'au sein d'un algorithme d'apprentissage par renforcement comme le Q-learning, il est donc nécessaire de distinguer la stratégie d'exploration simulée à chaque itération, qui est une stratégie aléatoire, et la meilleure stratégie courante, π_{Q_n}, qui elle est markovienne déterministe, et qui tend vers une stratégie optimale π^*.

22.3.4 La méthode des différence temporelles : TD(λ)

L'objectif de la méthode des différences temporelles (Sutton 1988) est d'évaluer au mieux la fonction de valeur V^π d'une stratégie π fixée, sur la base des seules transitions observées. Il s'agit donc une fois de plus d'un algorithme d'approximation stochastique d'une fonction solution d'une équation au point fixe (ici l'équation 22.4 pour le critère γ-pondéré). Bien qu'il ne s'agisse pas directement d'apprendre une stratégie optimale comme dans le Q-learning, il s'avère que le principe du TD(λ) est directement présent dans la définition de ces deux algorithmes. Son étude nous permet ainsi de mieux comprendre leur fonctionnement, et autorise leur généralisation comme à travers le Q(λ)-learning par exemple.

Nous avons vu de plus qu'il existait des méthodes de résolution de l'équation d'optimalité de Bellman qui réclamaient de calculer à chaque itération la fonction de valeur associée à la stratégie courante (algorithmes de *policy iteration*). Ces algorithmes de programmation dynamique nécessitent la connaissance des probabilité de transition et des fonctions de récompense. Ce n'est pas la cas pour la méthode des différence temporelles, d'où son intérêt pour la mise au point de méthodes directes en apprentissage par renforcement.

Nous nous plaçons ici pour des raisons de simplicité de l'exposé dans le cas d'un processus décisionnel de Markov et d'une stratégie π telle que la chaîne de Markov associée $p(s_{t+1} \mid s_t) = p(s_{t+1} \mid s_t, \pi(s_t))$ conduise pour tout état initial vers un état terminal T absorbant de récompense nul. On considère donc la critère total et l'on cherche à estimer $V(s) = E(\sum_0^\infty r_t \mid s_0 = s)$ à partir des seules observations (s_t, s_{t+1}, r_t). C'est le cas par exemple du problème du parking, puisque toute stratégie se termine au pire dans le parking souterrain.

Méthodes de Monte Carlo

L'approche indirecte de type *maximum de vraisemblance* qui consiste à estimer les paramètre $p()$ et $r()$ puis à calculer V en résolvant l'équation 22.4 (avec ici $\gamma = 1$) est généralement trop coûteuse, en temps et en espace.

On lui préfère classiquement l'approche dite de *Monte Carlo*, qui revient à simuler un grand nombre de trajectoires issues de chaque état s de S, et à estimer $V(s)$ en moyennant les coûts observés sur chacune de ces trajectoires. Soit ainsi $(s_0, s_1, \ldots, s_N)$ une trajectoire générée en suivant la probabilité de transition inconnue $p()$, et $(r_0, r_1, \ldots, r_{N-1})$ les récompenses observées (s_N est l'état terminal T de récompense nul).

Le principe de la méthode de Monte Carlo est de mettre à jour les N valeurs $V(s_k)$, $k = 0, \ldots, N-1$, selon :

$$V(s_k) \leftarrow V(s_k) + \alpha(s_k)(r_k + r_{k+1} + \cdots + r_{N-1} - V(s_k)) \tag{22.6}$$

avec les taux d'apprentissage $\alpha(s_k)$ tendant vers 0 au cours des itérations. La convergence presque sûre de cet algorithme vers la fonction V est assurée sous des hypothèses générales (Bertsekas and Tsitsiklis 1996).

Cette méthode est qualifiée d'*every-visit* car la valeur d'un état peut être mise à jour plusieurs fois le long d'une même trajectoire. Les termes d'erreur associés à chacune de ces mises à jours ne sont alors pas indépendants, entraînant un biais non nul dans l'estimation de la fonction V sur la base d'un nombre fini de trajectoires ((Bertsekas and Tsitsiklis 1996, page 190). Une solution simple consiste alors à ne mettre à jour la valeur $V(s)$ d'un état que lors de sa première rencontre le long de la trajectoire observée. Cela définit la méthode dite de *first-visit*, qui conduit à un estimateur non-biaisé de la fonction V. Il s'avère expérimentalement que l'erreur quadratique moyenne de la *first-visit* méthode tend à être inférieure à celle de l'*every-visit* méthode (Singh and Sutton 1996).

Les méthodes de Monte Carlo que nous venons de voir permettent donc d'estimer la fonction de valeur d'une stratégie π, en mettant à jour certaines de ses composantes à la fin de chaque trajectoire observée. Il est possible d'améliorer ces algorithmes en autorisant la mise à jour de la fonction de valeur non plus simplement à la fin de chaque trajectoire, mais à la suite de chaque transition du système.

La règle de mise à jour 22.6 de la fonction V peut être réécrite de la manière suivante (nous utilisons la propriété $V(s_N) = V(T) = 0$) :

$$\begin{aligned} V(s_k) \leftarrow V(s_k) + \alpha(s_k) \quad & \Big((r_k + V(s_{k+1}) - V(s_k)) \\ & + (r_{k+1} + V(s_{k+2}) - V(s_{k+1})) \\ & + \cdots \\ & + (r_{N-1} + V(s_N) - V(s_{N-1})) \Big) \end{aligned}$$

soit encore

$$V(s_k) \leftarrow V(s_k) + \alpha(s_k)(d_k + d_{k+1} + \cdots + d_{N-1}) \tag{22.7}$$

en définissant la différence temporelle d_k par

$$d_k = r_k + V(s_{k+1}) - V(s_k), \quad k = 0, \ldots, N-1.$$

Cette erreur d_k peut être interprétée en chaque état comme une estimation de la différence entre l'estimation courante $V(s_k)$ et l'estimation corrigée à un coup $r_k + V(s_{k+1})$. Son calcul est possible dès que la transition (s_k, s_{k+1}, r_k) a été observée, et cela conduit donc à une version « on-line » de la règle de mise à jour 22.7 où il n'est plus nécessaire d'attendre la fin de la trajectoire pour commencer à modifier les valeurs de V :

$$V(s_l) \leftarrow V(s_l) + \alpha(s_l)d_k, \quad l = 0, \ldots, k \tag{22.8}$$

dès que la transition (s_k, s_{k+1}, r_k) est simulée et l'erreur d_k calculée. Selon qu'une trajectoire puisse parcourir plusieurs fois le même état ou non, cette version « on-line » peut légèrement différer de l'algorithme original 22.7. Toutefois sa convergence presque sûre vers V reste valide. Là encore, une approche de type *first-visit* semble préférable en pratique (Singh and Sutton 1996).

Méthode de la programmation dynamique

Poursuivant la même démarche que celle ayant abouti à l'algorithme de *value iteration* 22.3 à partir de l'équation de Bellman, il est possible de dériver de 22.4 une nouvelle règle de mise à jour pour la fonction de valeur V d'une stratégie :

$$\begin{aligned} V(s_k) &\leftarrow V(s_k) + \alpha(s_k)(r_k + V(s_{k+1}) - V(s_k)) && (22.9)\\ &\leftarrow V(s_k) + \alpha(s_k)d_k && (22.10)\end{aligned}$$

en chaque point s_k de la trajectoire. La valeur de $V(s_k)$ est donc mise à jour uniquement à chaque passage en s_k, avec un terme d'erreur limité à un horizon d'un coup.

TD(λ)

L'originalité de la méthode TD(λ) est de proposer un compromis entre ces deux équations 22.7 et 22.10. Soit donc $\lambda \in [0, 1]$ un paramètre de pondération. Avec les mêmes notations que précédemment, l'algorithme de TD(λ) défini par Sutton (Sutton 1988) est le suivant :

$$V(s_k) \leftarrow V(s_k) + \alpha(s_k) \sum_{m=k}^{m=N-1} \lambda^{m-k} d_m, \quad k = 0, \ldots, N-1. \tag{22.11}$$

On peut essayer de mieux comprendre le rôle du coefficient λ en réécrivant l'équation 22.11 sous la forme

$$V(s_k) \leftarrow V(s_k) + \alpha(s_k)(z_k^\lambda - V(s_k)).$$

On a alors

$$\begin{aligned} z_k^\lambda &= V(s_k) + \sum_{m=k}^{m=N-1} \lambda^{m-k} d_m \\ &= V(s_k) + d_k + \lambda \sum_{m=k+1}^{m=N-1} \lambda^{m-k-1} d_m \\ &= V(s_k) + d_k + \lambda(z_{k+1}^\lambda - V(s_{k+1})) \\ &= V(s_k) + r_k + V(s_{k+1}) - V(s_k) + \lambda(z_{k+1}^\lambda - V(s_{k+1})) \\ &= r_k + (\lambda z_{k+1}^\lambda + (1-\lambda)V(s_{k+1})) \end{aligned}$$

Ainsi, le cas $\lambda = 1$ revient à prendre en compte le coût total observé de la trajectoire, comme dans la méthode de Monte Carlo, et le cas $\lambda = 0$ revient à ne considérer qu'un horizon de un coup, comme pour la méthode de la programmation dynamique.

Pour tout λ, les deux approches de type *first-vist* ou *every-visit* peuvent être considérée. De même, une version *on-line* de l'algorithme d'apprentissage TD(λ) décrit par l'équation 22.11 est possible :

$$V(s_l) \leftarrow V(s_l) + \alpha(s_l)\lambda^{k-l}d_k, \quad l = 0, \ldots, k \tag{22.12}$$

dès que la transition (s_k, s_{k+1}, r_k) est simulée et l'erreur d_k calculée.

La convergence presque sûre l'algorithme TD(λ) a été montré pour toute valeur de λ, en *on-line* ou *off-line*, sous les hypothèses classiques de visite en nombre infini de chaque état $s \in S$, et décroissance des α vers 0 à chaque itération n, telle que $\sum_n \alpha_n(s) = \infty$ et $\sum_n \alpha_n^2(s) < \infty$ (Jaakkola *et al.* 1994; Bertsekas and Tsitsiklis 1996).

L'application du TD(λ) pour l'évaluation d'une stratégie π selon le critère γ-pondéré entraîne certaines modifications des algorithmes standards 22.11 ou 22.12, qu'il est nécessaire de citer ici.

Un calcul en tout point semblable au cas $\gamma = 1$ conduit à une règle du type :

$$V(s_k) \leftarrow V(s_k) + \alpha(s_k) \sum_{m=k}^{m=\infty} (\gamma\lambda)^{m-k} d_m. \tag{22.13}$$

Il est alors clair que l'absence potentielle d'états finaux absorbants rend inadéquate un algorithme de type *off-line* ne mettant à jour la fonction de valeur V qu'à la fin de la trajectoire, car celle ci peut être de taille infinie. On définit donc une version *on-line* de 22.13, qui prend la nouvelle forme suivante :

$$V(s) \leftarrow V(s) + \alpha(s)z_n(s)d_n, \quad \forall s \in S \tag{22.14}$$

dès que la $n$ième transition (s_n, s_{n+1}, r_n) a été simulée et l'erreur d_n calculée. Le terme $z_n(s)$, dénommé trace d'éligibilité[2] se définit ainsi dans la version la plus proche de l'algorithme TD(λ) original :

Définition 22.6 – Trace d'éligibilité accumulative –

$$\begin{aligned} z_0(s) &= 0, \quad \forall s \in S \\ z_n(s) &= \begin{cases} \gamma\lambda z_{n-1}(s) & si\ s \neq s_n \\ \gamma\lambda z_{n-1}(s) + 1 & si\ s = s_n \end{cases} \end{aligned}$$

Ce coefficient d'éligibilité augmente donc sa valeur à chaque nouveau passage dans l'état associé, puis décroît exponentiellement au cours des itérations suivantes, jusqu'à un nouveau passage dans cet état (voir figure 22.3).

Dans certains cas une définition légèrement différente de la trace $z_n(s)$ semble conduire à une convergence plus rapide de la fonction de valeur V :

---

2 Traduit de l'anglais *eligibility trace*, ou encore *activity*.

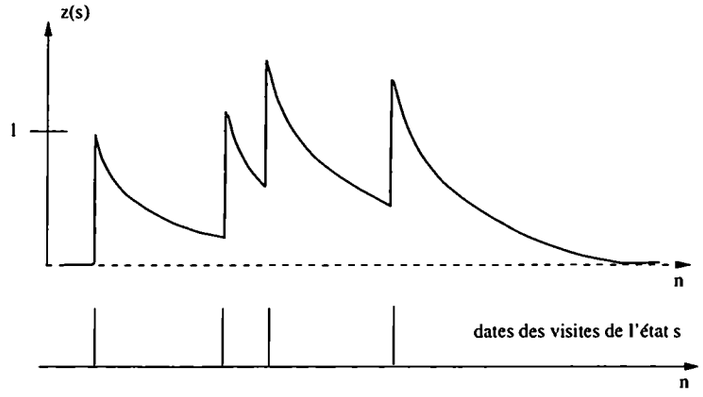


Figure 22.3 – Trace d'éligibilité cumulative

Définition 22.7 – Trace d'éligibilité avec réinitialisation –

$$z_0(s) = 0, \quad \forall s \in S$$
$$z_n(s) = \begin{cases} \gamma\lambda z_{n-1}(s) & si\ s \neq s_n \\ 1 & si\ s = s_n \end{cases}$$

La valeur de la trace est donc saturée à 1, comme le montre la figure 22.4.

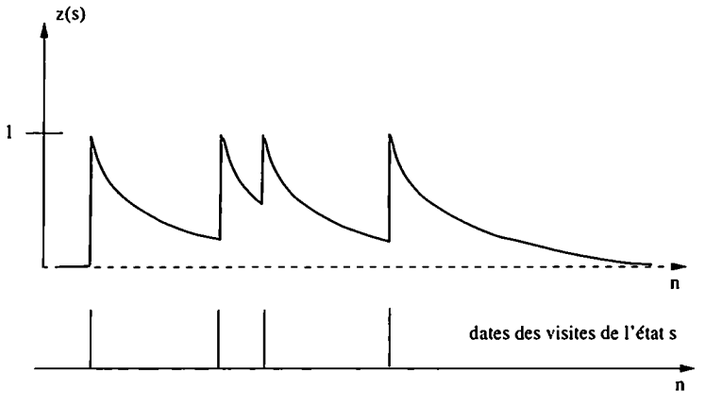


Figure 22.4 – Trace d'éligibilité avec réinitialisation

Une implémentation directe de TD(λ) basée sur la trace d'éligibilité n'est bien sûr pas efficace dès que la taille de l'espace d'état S devient trop grande. Une première solution approchée (Sutton and Barto 1998) consiste à forcer à 0 la valeur de toutes les traces $z_n(s) < \epsilon$, et donc à ne maintenir que les traces des états récemment visités (plus précisément, on cesse de maintenir un état dont la dernière visite remonte à plus de $\frac{\log(\epsilon)}{\log(\gamma\lambda)}$) transitions.

Une autre méthode approchée proposée par Cichosz (Cichosz 1995) sous le nom de *truncated temporal differences*, ou TTD(λ), revient à gérer un horizon glissant de taille m mémorisant les derniers états visités et à mettre à jour sur cette base à chaque itération n la valeur de l'état visité à l'itération $n - m$.

Il est à noter que l'effet du paramètre λ est encore mal compris, et sa détermination optimale pour un problème donné reste très empirique. Quelques travaux, dont (Singh and Dayan 1998), tendent à établir l'optimalité de valeurs intermédiaires de λ entre 0 et 1 pour des taux d'apprentissage constants. Dans le cas $\gamma < 1$, une étude asymptotique de TD(λ) pour des taux en $\frac{1}{n}$ montre au contraire que des vitesses optimales de convergence sont obtenues pour $\lambda = 1$ (Garcia and Serre 2000).

TD(λ) et Q-learning

TD(λ) peut être appliqué au problème de l'apprentissage par renforcement pour apprendre une stratégie optimale. Pour cela, il est nécessaire de coupler à TD(λ) un algorithme gérant l'évolution d'une suite de stratégies π_n. En effet, contrairement au Q-learning qui voit la suite Q_n converger vers Q^* sans nécessiter la présence en parallèle d'une suite de stratégies π_n, l'algorithme TD(λ) ne sait qu'apprendre la fonction de valeur d'une stratégie fixée. Dans Q-learning une telle suite de stratégies existe à travers π_{Q_n}, mais l'intérêt du Q-learning est justement que cette suite n'est qu'implicite. On retrouve donc ici un type d'opposition rencontrée en programmation dynamique entre *value iteration* et *policy iteration*.

Malgré cela, il s'avère que l'algorithme Q-learning intègre directement l'idée maîtresse de TD(λ) de considérer une erreur de différence temporelle. Si l'on reprend la règle de mise à jour du Q-learning

$$Q_{n+1}(s_n, a_n) = Q_n(s_n, a_n) + \alpha_n\{r_n + \gamma V_n(s'_n) - Q_n(s_n, a_n)\}$$

pour la transition observée (s_n, a_n, s'_n, r_n), et dans le cas où l'action a_n exécutée dans l'état s_n est l'action optimale pour Q_n, c'est à dire $a_n = \pi_{Q_n}(s_n) = \text{argmax}_b\, Q_n(s_n, b)$, on constate que le terme d'erreur employé est égal à

$$r_n + \gamma V_n(s'_n) - V_n(s_n)$$

qui est exactement celui de TD(0). Cela peut se généraliser à $\lambda > 0$, et une première approche pour tenter de coupler les méthodes TD(λ) et Q-learning est présenté dans l'algorithme SARSA(λ) (Rummery and Niranjan 1994). Cet algorithme reprend directement l'équation 22.14 en l'adaptant à une représentation par Q-valeur.

La trace d'éligibilité $z_n(s, a)$ est étendue aux couples état-action, et l'exploration de l'espace d'états est guidée par la dynamique (sauf lors de la rencontre avec un état terminal). La différence essentielle entre SARSA(λ) et Q-learning se situe au niveau de la définition du terme d'erreur d_n. Dans SARSA, le terme $\max_b Q_n(s'_n, b)$ est remplacé par $Q_n(s'_n, a'_n)$, ce qui n'est équivalent que lorsque l'on a choisi l'action $a'_n = \pi_{Q_n}(s'_n)$ (a'_n est une *greedy action*).

La prise en compte des cas où l'action optimale $\pi_{Q_n}(s'_n)$ n'a pas été sélectionnée conduit aux algorithmes $Q(\lambda)$ proposés par Watkins (voir (Sutton and Barto 1998)) et Peng (Peng and Williams 1994). La caractéristique du $Q(\lambda)$ de Watkins est de ne considérer un $\lambda > 0$ que le long des segments de trajectoires où la stratégie courante π_{Q_n} a été suivie. Les deux modifications relativement à SARSA(λ) concernent donc les règles de mise à jour de Q_n et de z_n :

Une limitation apparente de cette approche réside dans le fait que pour des stratégies d'apprentissage très exploratrices, les traces z_n sont très fréquemment remises à 0, et le comportement de Q(λ) est alors assez proche du Q-learning original. Le $Q(\lambda)$ de Peng est une réponse à ce problème. Il est aussi possible d'imaginer une application directe de TD(λ) au Q-learning en ne remettant pas la trace z_n à 0 lors du choix d'une action non-optimal. Il existe peu de résultats expérimentaux présentant cette approche (voir toutefois (Ndiaye 1999)) ni de comparaisons avec SARSA(λ) et Q(λ) autorisant de tirer des conclusions définitives sur le sujet. Toutefois, on peut montrer que la remise à zéro de la trace lors de l'exploration dans le Q(λ) de Watkins participe à la minimisation de la

Algorithme 17: SARSA(λ)

Initialiser(Q_0);
$z_0 \leftarrow 0$;
$s_0 \leftarrow$ ChoixEtat ;
$a_0 \leftarrow$ ChoixAction ;
pour $n \leftarrow 0$ **Jusqu'à** N_{tot} **faire**
  $(s'_n, r_n) \leftarrow$ Simuler(s_n, a_n);
  $a'_n \leftarrow$ ChoixAction ;
  { mise à jour de Q_n et z_n };
  début
    $d_n \leftarrow r_n + \gamma Q_n(s'_n, a'_n) - Q_n(s_n, a_n)$;
    $z_n(s_n, a_n) \leftarrow z_n(s_n, a_n) + 1$;
    pour $s \in S, a \in A$ **faire**
      $Q_{n+1}(s, a) \leftarrow Q_n(s, a) + \alpha_n(s, a) z_n(s, a) d_n$;
      $z_{n+1}(s, a) \leftarrow \gamma \lambda z_n(s, a)$;
  fin
  si s'_n *non absorbant* **alors** $s_{n+1} \leftarrow s'_n$ et $a_{n+1} \leftarrow a'_n$;
  sinon
    $s_{n+1} \leftarrow$ ChoixEtat ;
    $a_{n+1} \leftarrow$ ChoixAction ;
retourner Q_n;

variance asymptotique de l'estimation Q_n, conduisant ainsi à une vitesse de convergence optimale (Garcia and Serre 2001).

En conclusion, la seule véritable certitude issue de nombreuses applications est que la prise en compte des traces d'éligibilité avec un $\lambda > 0$ accélère la convergence de l'apprentissage en terme de nombre d'itérations. L'analyse en terme de temps de calcul est plus complexe, car les algorithmes prenant en compte une trace nécessitent beaucoup plus de calcul à chaque itération.

22.4 Apprentissage par renforcement et généralisation

Tous les algorithmes que nous venons de présenter, de type programmation dynamique ou apprentissage par renforcement, supposent une représentation discrète et finie des espaces d'états et d'actions S et A du problème. Pour un grand nombre d'applications ces conditions ne peuvent être vérifiées, principalement pour deux raisons :

- soit le problème est discret par nature, mais de trop grande taille : les algorithmes de programmation dynamique butent sur des contraintes de temps de calcul et de taille mémoire ; les algorithmes d'apprentissage par renforcement ne peuvent visiter tous les états en un temps raisonnable et donc ne proposent pas d'action optimale en certains états.
- soit le problème est à espace d'états ou d'actions continus, et les algorithmes ne peuvent être directement appliqués.

Algorithme 18: Le Q(λ) de Watkins

Initialiser(Q_0);
$z_0 \leftarrow 0$;
$s_0 \leftarrow$ ChoixEtat ;
$a_0 \leftarrow$ ChoixAction ;
pour $n \leftarrow 0$ **Jusqu'à** N_{tot} **faire**
  $(s'_n, r_n) \leftarrow$ Simuler(s_n, a_n);
  $a'_n \leftarrow$ ChoixAction ;
  { mise à jour de Q_n et z_n };
  début
    $d_n \leftarrow r_n + \gamma \max_b Q_n(s'_n, b) - Q_n(s_n, a_n)$;
    $z_n(s_n, a_n) \leftarrow z_n(s_n, a_n) + 1$;
    pour $s \in S, a \in A$ **faire**
      $Q_{n+1}(s,a) \leftarrow Q_n(s,a) + \alpha_n(s,a) z_n(s,a) d_n$;
      $z_{n+1}(s,a) \leftarrow \begin{cases} 0 & \text{si } a'_n \neq \pi_{Q_n}(s'_n) \\ \gamma\lambda z_n(s,a) & \text{si } a'_n = \pi_{Q_n}(s'_n) \end{cases}$;
  fin
  si s'_n *non absorbant* **alors** $s_{n+1} \leftarrow s'_n$ et $a_{n+1} \leftarrow a'_n$;
  sinon
    $s_{n+1} \leftarrow$ ChoixEtat ;
    $a_{n+1} \leftarrow$ ChoixAction ;
retourner Q_n;

Ainsi pour des problèmes de grande dimension ou à domaines continus, la simple représentation des fonctions de valeur V ou Q selon un tableau (*look-up table*) doit être dépassée, et il est alors nécessaire d'utiliser des techniques de *généralisation* permettant d'étendre l'apprentissage de la valeur d'une fonction d'un point à son voisinage.

Il existe heureusement de très nombreuses méthodes efficaces de généralisation à partir d'exemples, tirées de l'apprentissage automatique ou des statistiques, et la seule difficulté est de concevoir leur couplage avec les algorithmes d'apprentissage par renforcement.

22.4.1 Représentation paramétrée de la fonction de valeur

L'objectif final des méthodes de généralisation en apprentissage par renforcement est d'aboutir à une représentation paramétrée V_ξ de la fonction de valeur $V = V^\pi$ d'une stratégie π fixée (estimation), ou de $V = V^*$ associée à la stratégie optimale à apprendre (optimisation), et où ξ est un vecteur de paramètres de faible taille, classiquement élément de $\mathbb{R}^n$. L'apprentissage porte alors sur ξ :

$$\xi_{n+1} = \xi_n + \Delta(s_n, a_n, s_{n+1}, r_n)$$

Les propriétés recherchées pour une bonne paramétrisation d'une fonction de valeur sont

la capacité d'approximation : l'approximation de V par V_ξ induit généralement une perte d'optimalité intrinsèque à la paramétrisation retenue, car rien n'indique que la fonction de valeur recherchée V appartienne à l'ensemble des fonctions de valeur V_ξ représentables. Pour une paramétrisation donnée, cette erreur minimale peut être représentée par $\varepsilon = \min_\xi \|V - V_\xi\|$, pour une norme $\|.\|$ adaptée, et l'on recherche une paramétrisation qui minimise ε ;

la capacité de généralisation : il est clair qu'une paramétrisation optimale du point de vue de l'erreur d'approximation ε consiste (pour S discret) à poser pour ξ le vecteur V lui-même. Cela n'est bien sûr pas satisfaisant, car l'on recherche des ξ de faible dimension. Cela implique nécessairement une corrélation forte entre les valeurs de V_ξ : lors de l'apprentissage, les valeurs de plusieurs états seront simultanément modifiées avec la présentation d'un seul exemple ;

la simplicité de l'évaluation : selon la paramétrisation retenue, le calcul pour un état $s \in S$ de $V_\xi(s)$ peut être plus ou moins complexe. Alors que pour une représentation en tableau, il suffit simplement d'aller lire une valeur à une adresse donnée, certaines représentations nécessitent un grand nombre d'opérations arithmétiques, ralentissant d'autant la vitesse globale du processus d'apprentissage de la fonction de valeur optimale ;

l'efficacité de l'apprentissage : toutes les paramétrisations envisageables de V ne conduisent pas à la définition d'algorithmes d'apprentissage portant sur ξ présentant les mêmes propriétés de convergence. Souvent, la convergence n'est plus formellement établie. De même, le lien entre ξ et V_ξ est parfois tel que le principe de l'équation de Bellman n'est plus directement adaptable, et un algorithme d'apprentissage par renforcement d'une stratégie optimale ne peut être imaginé. Dans ce cas, il reste la possibilité d'adapter la paramétrisation en question directement à l'expression des stratégies π, et de chercher à approcher π^* par des techniques locales d'optimisation (voir section 22.4.4).

De nombreuses architectures d'approximation sont utilisées en apprentissage par renforcement. Elles se distinguent essentiellement par les règles d'apprentissage des paramètres ξ qu'il est possible de définir. Une première approche, la plus employée, consiste à retenir une représentation paramétrée V_ξ différentiable en ξ. Le vecteur ξ est typiquement un vecteur de paramètres réels instanciant une structure de représentation définie a priori. Une seconde approche, plus complexe, s'autorise l'apprentissage de la structure même de la paramétrisation, pour laquelle la notion de gradient n'est plus adaptée.

22.4.2 Les représentations différentiables

Pour une telle représentation, il est naturel d'utiliser des méthodes de type descente de gradient pour apprendre le vecteur de paramètres ξ approchant une fonction de valeur V :

$$\xi_{n+1} = \xi_n + \alpha_n(R_n - V_{\xi_n}(s_n))\nabla_{\xi_n} V_{\xi_n}(s_n) \qquad (22.15)$$

où R_n est l'estimation directe tirée de l'expérience de la valeur de V en s_n. Comme nous l'avons vu dans la section 22.3.4, R_n peut être calculée par une méthode de Monte Carlo, par Programmation Dynamique ou par TD(λ). Seule Monte Carlo assure que R_n soit une estimation non-biaisée de V, et donc que la règle de mise à jour 22.15 converge vers

un optimum local pour la norme quadratique $\varepsilon(\xi) = \sum_{s \in S}(V(s) - V_\xi(s))^2$. Toutefois, cette propriété est aussi vérifiée pour TD(λ) dans certains cas particuliers.

Nous présentons succinctement ci-dessous [3] les principales méthodes basées sur l'équation 22.15, qui se décomposent selon les méthodes linéaires et non-linéaires. Pour chacune d'entre elles, une traduction en terme de Q-valeurs conduit à des algorithmes de type Q-learning, ou $Q(\lambda)$ selon le choix de R_n retenu.

Approximations linéaires

On suppose ici que la fonction de valeur V à estimer ou optimiser peut être approchée par une combinaison linéaire

$$V_\xi(s) = \xi(1)\psi_1(s) + \cdots \xi(K)\psi_K(s)$$

où les $\xi(i)$ sont les K paramètres à apprendre, et où les $\psi_i()$ sont K fonctions de S dans $\mathbb{R}$, qui représentent des caractéristiques essentielles de l'état pour le problème considéré.

Pour une telle représentation linéaire, le gradient $\nabla_{\xi_n} V_{\xi_n}(s_n)$ est égal au vecteur de composantes $\psi_i(s_n)$, ce qui conduit à la règle de mise à jour :

$$\xi_{n+1}(i) = \xi_n(i) + \alpha_n(R_n - V_{\xi_n}(s_n))\psi_i(s_n), \quad \forall i = 1, \ldots, K. \tag{22.16}$$

Une propriété importante de la représentation linéaire est l'existence d'un optimum unique ξ^* pour l'erreur quadratique $\varepsilon(\xi)$. De plus, on montre que la méthode du gradient couplée à TD(λ) converge nécessairement dans le cas linéaire (mais non forcément vers ξ^*) (Bertsekas and Tsitsiklis 1996) (sec. 6.3.3).

La définition de ces fonctions caractéristiques, ou *features*, est l'étape principale de la paramétrisation. Elles peuvent être obtenues de différentes manières, en exploitant la géométrie du domaine, ou la connaissance a priori que l'on peut avoir sur le problème de décision. Citons ainsi :

Les fonctions de voisinage

L'idée principale des fonctions de voisinage est de définir K régions $\mathcal{R}_i$ qui forment ensemble une couverture de S, et de poser pour $i = 1, \ldots, K$ $\psi_i() = \mathbb{1}_{\mathcal{R}_i}()$ la fonction indicatrice de $\mathcal{R}_i$

$$\forall s \in S \quad \psi_i(s) = \begin{cases} 1 & \text{si } s \in \mathcal{R}_i, \\ 0 & \text{sinon.} \end{cases}$$

Toutes les méthodes de recouvrement sont possibles, les deux plus connues étant celles basées sur des partitionnements et sur des états représentatifs.

Deux méthodes à base de partition sont bien connues. La première, connue sous le nom de *state aggregation* (regroupement d'états), est utilisée pour S discret de grande

3 Voir (Bertsekas and Tsitsiklis 1996; Sutton and Barto 1998) pour une analyse approfondie des ces différentes méthodes.

taille, et consiste à regrouper les états de S en K régions, et à associer une valeur à chacune de ces régions (Tsitsiklis and Roy 1996). Ici, un seul ψ_i peut donc être non nul à la fois. La fonction V est ainsi approchée par une fonction constante par morceau sur les $\mathcal{R}_i$. La convergence d'une adaptation du Q-learning à cette représentation est assurée (Bertsekas and Tsitsiklis 1996, § 6.7), avec une majoration de l'écart à la fonction de valeur optimale.

La seconde approche, introduite par Albus (Albus 1975) sous le terme de méthode des CMAC (*cerebellar model articulator controller*) et a été très largement utilisée en apprentissage par renforcement (Santamaría *et al.* 1996; Sutton 1995; Watkins 1989), particulièrement pour des espaces continus ou de très grande taille. Son principe est de définir C partitions (ou grilles) de S, décalées géométriquement les unes par rapport aux autres, et d'associer à chaque région i de chaque grille j une valeur ω_i^j. La valeur d'un état s de S est alors calculée comme la somme des C régions de chaque grille qui le contiennent (voir figure 22.5). Si chaque grille j contient N_j régions, la taille du vecteur de paramètres $\xi = (\omega_1^1, \ldots, \omega_{N_C}^C)^T$ est alors $K = \sum_{j=1}^{C} N_j$.

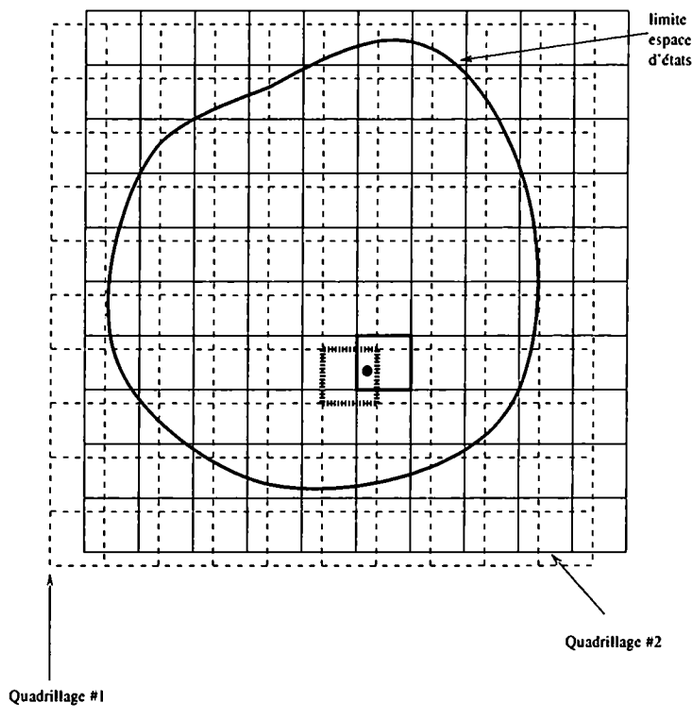


Figure 22.5 – CMAC avec $C = 2$ et $N = 100$

Les méthodes à base d'états représentatifs consistent à placer K points s_i dans S, et à définir autour de chacun de ces centres une région dont l'exemple le plus simple est une sphère de rayon R. Cela modélise ainsi très naturellement le concept de proximité à des exemples types (figure 22.6).

La généralisation de cette dernière méthode consiste a associer une valeur réelle dans l'intervalle $[0, 1]$ fonction de la distance au centre. Les fonctions de voisinage nommées *radial basis functions*, ou RBF, sont ainsi définies par

$$\forall s \in S \quad \psi_i(s) = \exp(-\frac{\|s - s_i\|}{2{\sigma_i}^2})$$

où σ_i est un paramètre fixé associé à s_i. L'avantage des RBF sur les fonctions de voisinages binaires est qu'elles sont différentiables relativement à leurs paramètres s_i et σ_i, et qu'elles

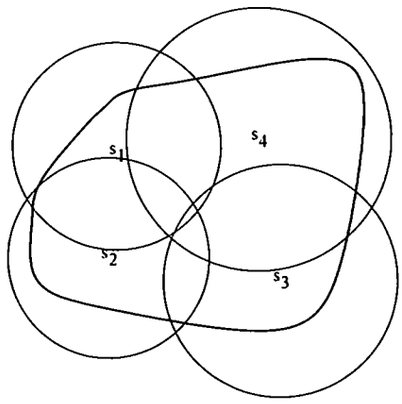


Figure 22.6 – Régions centrées circulaires avec $K = 4$

conduisent ainsi dans le cadre des approximations non-linéaires à des méthodes efficaces permettant d'apprendre en plus de ξ les meilleurs paramètres s_i et σ_i.

Une autre méthode à base d'états représentatifs (Bertsekas and Tsitsiklis 1996, exemple 6.16) consiste à utiliser une triangularisation de l'espace d'état $S \subset \mathbb{R}^n$ en un ensemble de points s_i, et à définir la fonction de valeur $V_\xi(s)$ en tout point s comme la combinaison linéaire

$$V_\xi(s) = \lambda_{s_{i_0}}(s)V_\xi(s_{i_0}) + \cdots \lambda_{s_{i_n}}(s)V_\xi(s_{i_n})$$

où les $V_\xi(s_{i_j})$ sont paramètres à apprendre, et où les $\lambda_{s_{i_j}}(s)$ sont les coordonnées barycentriques du point s par rapport aux $n+1$ points $s_{i_0}, \ldots, s_{i_n}$ du simplexe qui le contient. Cette méthode a été utilisée dans (Munos 1996) pour résoudre par Q-learning les problèmes de contrôle à temps continu.

Les fonctions coordonnées

Le principe est ici de projeter l'espace S de grande dimension dans un sous-espace de $\mathbb{R}^K$, en définissant pour cela K nouvelles coordonnées qui résument au mieux les propriétés principales d'un état s relativement à la tâche à apprendre. Ces nouvelles fonctions coordonnées sont typiquement élaborées après une expertise du problème, et chacune d'entre elles est généralement calculable à partir de la valeur d'un petit nombre de variables d'états de S.

Les fonctions heuristiques

Toujours afin d'exploiter au mieux toute connaissance a priori portant sur la résolution du problème, une dernière approche consiste à définir les fonctions $\psi_i()$ comme les fonctions de valeur associées à des stratégies obtenues par expertise ou par une première résolution approchée. Ainsi, si pour le problème à résoudre il est possible de définir K stratégies distinctes $\pi_1, \pi_2, \ldots, \pi_K$ ayant un comportement intéressant en certains régions de S, on pose

$$V_\xi(s) = \xi(1)V_{\pi_1}(s) + \cdots + \xi(K)V_{\pi_K}(s)$$

où V_{π_i} est la fonction de valeur de π_i estimée par simulation ou calculée si un modèle du processus est disponible. Ces fonctions V_{π_i} peuvent elles-mêmes être paramétrées (Bertsekas and Tsitsiklis 1996, § 3.1.4).

Approximations non-linéaires : le perceptron multi-couches

Le perceptron multi-couches est sans doute l'architecture non-linéaire la plus employée en apprentissage par renforcement. Sa forme la plus simple est définie par deux couches linéaires encadrant une couche cachée à base de fonctions sigmoïdes (figure 22.7).

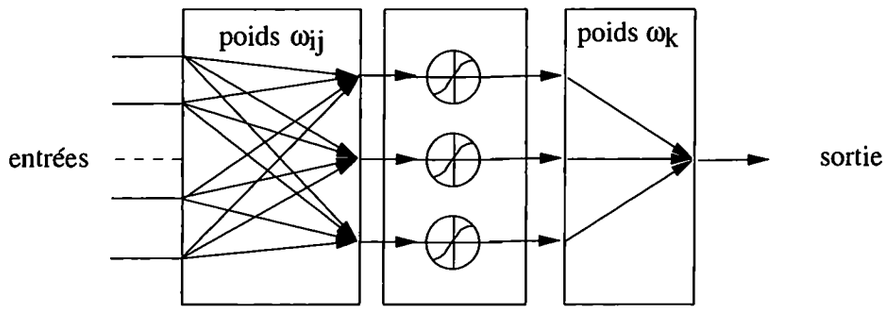


Figure 22.7 – Perceptron à une couche cachée

Pour un état s défini par ses variables d'état $(s(1), \ldots, s(i), \ldots)$, la fonction de valeur $V(s)$ est alors approchée par

$$V_\xi(s) = \sum_k \omega_k \, \sigma(\sum_i \omega_{k,i} s(i))$$

où $\sigma()$, nommée fonction sigmoïde, est monotone croissante, dérivable, et accepte deux limites finies en $-\infty$ et $+\infty$. Un exemple d'une telle fonction est donné par

$$\sigma(x) = \frac{1}{1 + \exp(-x)}.$$

Les perceptrons multi-couches sont une généralisation de ce principe avec des réseaux où s'enchaînent couches linéaires et couches sigmoïdales.

Le vecteur ξ des paramètres de ce perceptron est constitué des poids ω_i, ω_{jk}, ... du réseau. L'apprentissage de ces poids est réalisé à partir de la règle de gradient 22.15. Le calcul du terme de gradient $\nabla_{\xi_n} V_{\xi_n}$ est effectué par des techniques de rétro-propagation au sein du réseau, comme nous l'avons vu au chapitre 20 (dans le cas d'une unique couche cachée, ce gradient peut être directement calculé analytiquement).

On trouvera dans (Langlois 1992) et (Rummery and Niranjan 1994) l'étude de plusieurs algorithmes d'apprentissage par renforcement couplant Q-learning et TD(λ) avec des fonctions d'approximation à base de perceptrons multi-couches.

22.4.3 L'apprentissage de structures d'approximation

Les méthodes comme le regroupement d'états ou les CMAC reviennent à discrétiser l'espace d'états initial S selon un pas de discrétisation fixé a priori, et à apprendre sur les régions ainsi définies une valeur moyenne de la fonction V^*. Le choix optimal de ce pas de discrétisation est un problème difficile, car fortement lié à la fonction V^* à apprendre : sur certaines régions la fonction V^* est pratiquement constante, autorisant ainsi un pas très large, alors qu'en d'autres régions deux états très proches conduisent à des résultats très différents, nécessitant ainsi un pas très fin.

Il apparaît donc nécessaire d'utiliser une discrétisation de l'espace adaptée au problème à résoudre, et dans le cas de l'apprentissage par renforcement cette discrétisation à résolution spatiale variable ne peut qu'être apprise. Quelques algorithmes d'apprentissage par renforcement se sont attaqués à ce problème, en se basant sur une représentation par arbre de régression.

Méthodes à base d'arbres de régression

Les arbres de régression généralisent les arbres de décision à la prise en compte de variables réelles. Les arbres de décision ont initialement été introduits pour résoudre des problèmes d'estimation de fonctions entre espaces $\mathcal{X}$ et $\mathcal{Y}$ décrits par des attributs binaires ou multivalués. Le nœud interne d'un arbre de décision correspond à un test particulier sur la valeur d'un attribut, et les branches issues de ce nœud sont étiquetées par les différentes valeurs du test. Pour un exemple donné $x \in \mathcal{X}$, le calcul de la sortie de l'arbre de décision est simple. En partant de sa racine, on parcourt l'arbre suivant les branches correspondants aux réponses aux tests rencontrés, et ce jusqu'à une feuille terminale. Chaque feuille de l'arbre spécifie alors une valeur pour $\mathcal{Y}$.

Les arbres de régression utilisés en apprentissage par renforcement reçoivent en entrée les composantes binaires, multivaluées ou réelles $(s(1), \ldots, s(i), \ldots)$ de l'état courant s, et génèrent en sortie une estimation de la fonction de valeur recherchée $V(s)$ (on peut aussi chercher à estimer $Q(s, a)$).

Proposé par Moore (Moore 1993), *Parti-game* est un algorithme qui génère une discrétisation à résolution variable pour un domaine continu à dynamique déterministe, sur la base de données observées. Le principe général est d'associer la modification de la discrétisation courante aux résultats obtenues en suivant une stratégie apprise à partir de cette discrétisation. La discrétisation initiale ne contient qu'une unique classe représentant S, qui est alors itérativement scindée en deux.

Les algorithmes G (Chapman and Kaelbling 1991) et *U Tree* (McCallum 1995) approfondissent ce même schéma de discrétisation adaptative dans le cadre d'une représentation propositionnelle de l'état et des actions. Enfin, une généralisation de ces travaux aux domaines continus a été proposée dans l'algorithme *Continuous U Tree* (Uther and Veloso 1998).

Méthodes à base de règles

D'autres travaux relatifs à l'apprentissage de structures se sont portés sur le problème de la représentation des stratégies apprises sous formes de règles de décision aisément communicables.

Dans (Grefenstette *et al.* 1990) et (Ndiaye 1999) ces règles ont pour format

$$\begin{array}{ll} \mathcal{R} : & \text{poids } \omega, \\ \text{SI} & s_1 \in S^1 \wedge \cdots \wedge s_p \in S^p \\ \text{ALORS} & a_1 \in A^1 \wedge \cdots \wedge a_q \in A^q \end{array}$$

où les s_i et les a_j sont les coordonnées de l'état s et de l'action a, S^i et A^j des intervalles $[s_{1debut}; s_{1fin}]$ et $[a_{1debut}; a_{1fin}]$, et où ω est un poids associé à la règle. Une stratégie est alors représentée comme un ensemble de règles de ce type. Pour un état s plusieurs règles peuvent être actives, et la règle retenue est celle de poids ω maximal. L'action est alors tirée aléatoirement dans le pavé défini par les A^j.

L'apprentissage par renforcement de stratégies représentées de la sorte revient d'une part à apprendre les structures des règles que sont les intervalles $[s_{idebut}; s_{ifin}]$ et $[a_{jdebut}; a_{jfin}]$, et d'autre part l'ensemble des poids ω.

Dans (Grefenstette *et al.* 1990), cet apprentissage ne se fait pas selon le principe de l'apprentissage par renforcement, mais est plutôt abordé comme un problème d'optimisation stochastique, traité ici par des techniques d'algorithmes génétiques. Un individu est donc un ensemble de règles, l'évaluation d'une stratégie est faite par simulation, et l'algorithme génétique a pour tâche de générer une population de stratégies de plus en plus efficaces.

Dans (Ndiaye 1999), la démarche retenue reste plus proche des algorithmes classiques d'apprentissage par renforcement. Pour une structure fixée de pavés $DS = S^1 \times \cdots \times S^p$ et $DA = A^1 \times \cdots \times A^q$ recouvrant S et A, une fonction de valeur $R(DS, DA)$ est apprise par R-learning, et le gain moyen résultant ρ définit l'évaluation de la structure. Un algorithme génétique est alors couplé à cet apprentissage par renforcement pour optimiser les pavés DS et DA. En fin d'apprentissage, on choisit au sein de la meilleure structure pour chaque pavé DS le pavé DA qui maximise $R(DS, DA)$. Le poids ω de cette règle $DS \rightarrow DA$ est posé égal à $R(DS, DA)$.

Une représentation un peu différente des règles de décision, à base de modélisation flou, est proposée dans (Glorennec 94; Jouffe 1997; Berenji 94), à travers un algorithme nommé *fuzzy* Q-learning. Il s'agit d'apprendre des règles de décision floues de la forme :

$$\begin{array}{ll} \mathcal{R} : \text{SI} & s_1 \text{ est } \tilde{S}^1 \wedge \cdots \wedge s_p \text{ est } \tilde{S}^p \\ \text{ALORS} & a \text{ est } \tilde{A} \end{array}$$

où les $\tilde{S}^i$ et $\tilde{A}$ sont des ensembles flous.

Application aux PDM partiellement observables

L'intérêt de l'apprentissage en ligne d'architectures d'approximation ne se limite pas aux processus décisionnels markoviens de grande taille. Ainsi, dans le cadre des PDM partiellement observables, on a pu mettre en évidence deux types de problèmes qui gagnent à être abordés selon une approche basée sur l'apprentissage d'une structure d'approximation non prédéfinie (Whitehead and Lin 1995) :

- les méthodes à « représentations consistantes » : le vecteur d'observation est potentiellement très grand, et si le système peut-être considéré comme markovien, sa taille le rend intraitable. Il est alors intéressant d'optimiser en ligne le *contrôle de la perception*, selon un principe d'élimination des états internes non-markoviens, d'où la nécessité d'une représentation dynamique de la fonction de valeur ;
- les méthodes « à mémoire » : dans ce cas, le vecteur d'observation n'est *pas assez informatif* sur l'état courant s, et les informations passées peuvent être utilisées, sous la forme par exemple d'un réseau de neurones récurrent ou d'une chaîne de Markov cachée (McCallum 1993; Dutech 1999).

22.4.4 Méthodes de gradient dans l'espace des stratégies

L'ensemble des méthodes qui viennent d'être présentées tentent d'approcher la fonction de valeur optimale du problème, pour en dériver une stratégie proche de la stratégie optimale. Il existe toutefois des problèmes décisionnels de Markov tels que pour tout $\varepsilon > 0$, on puisse trouver une fonction de valeur approchée V avec $\|V - V^*\| = \varepsilon$ et pourtant $\|V^\pi - V^*\| = \frac{2\gamma\varepsilon}{1-\gamma}$ où π est la stratégie obtenue à partir de V. Ainsi, la simple approximation de V^* pour γ proche de 1 peut être insuffisante.

Il existe une approche alternative basée sur le principe de l'itération de la stratégie en programmation dynamique. L'idée consiste à considérer des stratégies paramétrées et à rechercher directement selon des méthodes de gradient les valeurs des paramètres qui maximisent le critère en espérance retenu. On peut ainsi imaginer dans l'exemple du parking des stratégies du type « tant que je suis au-delà d'une distance d du magasin je continue, ensuite je prends la première place libre ». Le seul paramètre à optimiser est d.

La plupart des algorithmes proposés se placent toutefois dans l'espace des stratégies paramétrées aléatoires, qui à toute paire (s, a) associent la probabilité $\pi_\theta(s, a)$ de choisir l'action a dans l'état s. Il est alors possible lier le gradient du critère à la fonction de valeur de la stratégie courante (Konda and Tsitsiklis 1999). Ce type de relation est à la base des différentes méthodes existantes.

On trouvera dans (Williams 1992; Baird and Moore 1999; Sutton *et al.* 1999; Baxter and Bartlett 2000) plusieurs algorithmes de gradient exploitant cette propriété. Les premiers résultats expérimentaux semblent démontrer la grande efficacité de ce type d'approche.

22.5 Conclusion

L'apprentissage par renforcement est devenu en quinze ans une des disciplines les plus actives de la communauté Intelligence Artificielle. À l'interface entre apprentissage, automatique, sciences cognitives, l'apprentissage par renforcement utilise des techniques variées pour aborder le problème de l'acquisition d'un comportement optimal dans un environnement incertain et dynamique.

Les méthodes de l'apprentissage par renforcement présentées dans ce chapitre sont adaptées pour les

- les problèmes de décisions séquentielles,
- à espaces d'états de grande taille,

– avec des récompenses immédiates ou retardées,
– et lorsqu'un modèle de simulation est disponible.

Dans ce cadre là, de nombreuse applications de l'apprentissage par renforcement ont été développées.

Les jeux ont bien sûr fait l'objet des premières applications en Intelligence Artificielle du principe de renforcement (voir Chapitre 15). L'exemple le plus classique est le programme de dames américaines *Checkers*, de Arthur Samuel (Samuel 1959) dans les années 50.

D'autres exemples ont suivi, comme le programme d'Othello BILL du CMU , ou le programme de *BackGammon* TD-GAMMON de Tesauro (Tesauro 1992; 1994) , qui ont tous deux atteint un niveau de classe mondiale en intégrant des techniques d'apprentissage par renforcement.

Aujourd'hui, d'autres problèmes de jeux plus complexes sont abordés, comme les échecs ou le Go. KNIGHTCAP (Baxter *et al.* 1998) est un programme d'échecs qui combine TD(λ) avec une recherche minimax. L'apprentissage de KNIGHTCAP en 600 parties a fait passer sa force de jeux de 1650 à 2150 ELO, l'adjonction d'une bibliothèque d'ouvertures amenant finalement son niveau à 2500 ELO. On trouve de même dans (Schraudolph *et al.* 1994) une application de TD(λ) à l'évaluation d'une position pour le Go, avec des résultats moins convaincants pour l'instant.

L'autre grand classique des applications de l'apprentissage par renforcement est la robotique, qui a elle-même développé le concept d'agent réalisant la boucle perception / raisonnement / action, et où l'apprentissage par renforcement est de plus en plus présent comme réponse au problème de l'autonomie décisionnelle. Citons ici quelques exemples :

– les robots jongleur, balanciers,... (Schaal and Atkeson 1994; Kimura and Kobayashi 1997)
– l'apprentissage de tâches d'assemblage (Asada and Liu 1991)
– la robotique mobile, la navigation (Mahadevan and Connell 1992; Pastor 1998).

Enfin, plus récentes, les applications industrielles portant sur l'optimisation de la conduite de systèmes de production, de communication, etc. sont de plus en plus nombreuses. On y intègre généralement les méthodes d'approximation que nous avons vues à la section 22.4 car la taille des problèmes considérés est souvent très importante. Nous listons ci-dessous quelques exemples récents :

– le remplissage temps-réel de containers (Kaelbling *et al.* 1996)
– la maintenance optimale de machines (Mahadevan *et al.* 1997)
– l'ordonnancement d'ascenseurs (Crites and Barto 1996)
– le *job-shop scheduling* (Zhang and Dietterich 1995)
– l'allocation de fréquences (Bertsekas and Tsitsiklis 1996, exemple 2.5)
– le routage de paquets (Boyan and Littman 1993)
– la conduite de cultures agricoles (Ndiaye 1999)
– la maintenance d'une constellation de satellites (Garcia *et al.* 2001).

Terminons ce chapitre en évoquant une dernière ouverture de l'apprentissage par renforcement à une discipline récente elle aussi, les multi-agents. Michael Littman (Littman 1994) a initié ce domaine en s'appuyant sur la théorie des jeux à deux joueurs à somme nulle. Cette approche a été approfondie dans (Hu and Wellman 1998) aux jeux markoviens à somme quelconque, dont la solution repose sur la notion d'équilibre de Nash.

On trouvera dans (Littman 2001) une présentation étendue de cette problématique avec quelques premiers algorithmes et résultats théoriques de convergence.

Partie VI

Conclusion

Gaily bedight, A gallant Knight,
In sunshine and in shadow,
Had journeyed long,
Singing a song,
In search of Eldorado.

But he grew old —
This knight so bold —
And o'er his heart a shadow
Fell as he found
No spot of ground
That looked like Eldorado.

And, as his strength
Failed him at length,
He met a pilgrim shadow —
"Shadow," said he,
"Where can it be —
This land of Eldorado?"

"Over the Mountains
Of the moon,
Down the Valley of the Shadow,
Ride, boldly, ride,"
The shade replied, —
"If you seek for Eldorado!"

— *E. A. Poe*

CHAPITRE 23

Conclusion

Jean-Marc Alliot

Les présentations de certains domaines de l'intelligence artificielle ressemblent bien souvent aux menus de mauvais restaurants : on vous propose une Parmentière en Robe Légère, et l'on vous sert une patate mal bouillie avec de la margarine.

Nous espérons avoir, au cours de ces chapitres, quelque peu démythifiée l'IA. Que reste-t-il donc des rêves des pionniers, sinon des techniques mathématiques de recherche dans des espaces d'états, des techniques de représentation de connaissances ou des méthodes d'optimisation ? Peut-on au moins dire que nous avons avancé dans la voie qui conduit à une véritable intelligence artificielle ? Nous avons, tout au long de ces pages, présenté l'intelligence artificielle comme une science de l'ingénieur ; nous nous sommes restreints à des résultats et des algorithmes précis et nous nous sommes efforcés de présenter sans parti-pris les différents travaux faits en intelligence artificielle ; nous avons tenté de montrer pourquoi il s'agissait d'une discipline scientifique intégrée dans le cadre de cette science générale qui est en train de naître : l'informatique. Pourtant l'IA est une discipline qui est le sujet d'une importante polémique : certains chercheurs y sont outrancièrement favorables, d'autres résolument opposés, et ils ont déclenché ce qui ressemble presque à une guerre de religion, guerre qui s'est étendue à la psychologie et à la philosophie[1].

À partir d'un bilan de ce qu'est l'IA aujourd'hui, nous allons tenter de faire une analyse de ces positions, essayer de dégager les raisons historiques qui ont pu amener de semblables attitudes, et dire quelle est à notre avis l'attitude à avoir par rapport à un domaine présenté tantôt comme l'Antéchrist[2] tentant de remplacer l'homme par des machines, et tantôt comme un nouveau Messie venant résoudre nos problèmes et nos contradictions.

---

1 Cet aspect quasi-religieux est parfaitement illustré par une phrase de Jacques Pitrat qui, dans la postface du livre de Dreyfus (Dreyfus 1984), écrivait : « Il n'y a pas actuellement d'argument scientifique permettant de trancher ni pour, ni contre, la réalisation d'intelligences artificielles [...] C'est une question de croyance. Monsieur Dreyfus n'y croit pas, j'y crois... »

2 Ainsi, un prédicateur américain écrit (Hutchings 1992) : « Make no mistake, humanists see salvation in the computer. Computer scientists contend that within less than a decade the computer will control mankind's total being. This awesome power that is destined to guide the destiny of the human race is causing many experts to wonder if man is not now in the process of creating his own god [...] More than nineteen hundred-years ago the Apostle John wrote in Revelation 13 that the day would come when all the world would worship the image of the Beast that would command all to work, buy, and sell through a system of marks and numbers. According to Fifth Generation computer experts, that day may be less than a decade away [...] The Fifth Generation computer will, in effect, be a living entity. It will reproduce itself and program itself, and theoretically, one super computer could indeed

23.1 Les critiques

C'est vers le milieu des années soixante qu'un certain nombre de scientifiques commencèrent à être irrités du tapage fait autour de l'IA. Il faut dire que nombre de gens qui s'occupaient alors d'IA faisaient parfois preuve d'un optimisme (volontairement ?) exagéré.

23.1.1 Les erreurs de jugement

En 1968, un jeune maître international d'échecs écossais, David Levy, lançait un défi aux spécialistes de l'IA : il pariait 250 livres sterling qu'aucune machine ne le battrait sur un match en 6 parties avant 1978. Ce défi faisait suite au texte de Simon prédisant un programme au niveau d'un champion du monde en 1968. Levy prolongeait donc le défi de Simon de 10 ans.

En fait, Levy, excellent joueur d'échecs lui-même et bon informaticien, ne prenait pas beaucoup de risques. Il remportera facilement son match en 1978 contre Chess 4.3. Après une nulle dans la première partie, il remporta les deux suivantes. Lors de la quatrième, il tenta de jouer un jeu purement tactique et perdit. Il revint alors à un jeu plus classique et remporta aisément la cinquième partie et le match. Sur les quatre personnes qui avaient accepté le pari, trois d'entre elles payèrent les 250 livres (John McCarthy, Donald Michie, Seymour Papert) et Levy attendait toujours le chèque de la quatrième (Kozdrowicky) en 1991.

Levy reconduira le pari (avec une mise de 5000 dollars) sur 5 ans et gagnera de nouveau en 1983. Son pari tiendra en fait jusqu'en 1989, et le programme (DEEP THOUGHT) qui le battra sera très loin dans sa conception de ceux imaginés par les spécialistes d'IA en 1958, puisqu'il s'agit avant tout d'un programme de force brute[3].

Nombreux sont les exemples de textes des années 55-65 qui annonçaient monts et merveilles pour les dix années à venir, et dont le dixième n'est même pas effectué aujourd'hui. Ainsi, en 1957, Simon et Newell écrivaient (Newell *et al.* 1957) :

> « Je ne saurais mieux me résumer qu'en disant qu'il existe désormais des machines capables de penser, d'apprendre et de créer. »

---

control the total activity of every human being on planet Earth [...] Computers will be used to confuse, to delude mankind, because Satan is the author of confusion. Satan's evil, super genius linked to human technology will be a marriage truly made in hell. »

Traduction : « Ne vous y trompez pas, les humanistes considèrent l'ordinateur comme notre Sauveur. Les informaticiens prétendent que dans moins de dix ans l'ordinateur contrôlera le devenir de l'humanité. Ce pouvoir effroyable, qui a pour but de guider la destinée de la race humaine, fait que de nombreux experts se demandent si l'homme n'est pas en train de créer son propre Dieu [...] Il y a plus de dix-neuf siècles, l'apôtre Jean écrivit dans l'Apocalypse, chapitre XIII, que le jour viendra où le monde entier adorera l'image de la Bête qui commandera à tous de travailler, d'acheter et de vendre à travers un système de signes et de nombres. Selon les experts du projet Cinquième Génération, ce jour est peut-être à moins de dix ans de nous [...] L'ordinateur de la Cinquième Génération sera, en effet, une entité vivante. Il se reproduira, se reprogrammera, et théoriquement, un super-ordinateur pourrait, en fait, contrôler toute l'activité de chaque être humain sur la planète Terre [...] Les ordinateurs seront utilisés pour semer la confusion, pour tromper l'humanité, parce que Satan est l'auteur du chaos. Le mariage du génie maléfique de Satan et de la technologie humaine sera véritablement célébré en Enfer. ».

3 Un excellent article retraçant l'évolution des confrontations homme-machine est (Kopec 1990).

Ceci devrait nous encourager à relativiser bon nombre d'articles ou de textes qui continuent à être publiés aujourd'hui[4].

23.1.2 Concepts et modèles : le réductionnisme

Dans (Winston 1984), Patrick Winston écrit :

> « Confronté à un exemple, le programme construit une représentation structurelle de la scène sous la forme d'un réseau sémantique où les nœuds représentent les composants et les lignes les relations entre les composants. Après qu'on lui ait dit qu'il s'agissait d'une arche, le programme enregistre cette description comme étant une arche. »

Il faut immédiatement remarquer deux choses ; tout d'abord, l'auteur prête à son programme des actions (apprendre) qui ont pour effet de renforcer l'aspect intelligent du programme. Certains auteurs (Schank par exemple) estiment en effet que leur programme manifeste une véritable compréhension du sujet, position violemment critiquée par d'autres. (Ainsi, le philosophe John Searle (Searle 1987) a écrit : « Je fais un distinguo entre l'*IA forte* et l'*IA faible*. L' IA forte prétend que les programmes d'ordinateurs ont des états cognitifs au sens littéral et que, donc, les programmes sont des théories psychologiques. Je prétends que l'IA forte est fausse... ».)

Le point moins évident est la confusion que fait P. Winston entre le *concept* de l'arche et le *modèle* de l'arche. Il est évident que même en acceptant les termes utilisés, le programme n'aura, en aucune façon, appris le concept d'arche, mais seulement un modèle géométrique de ce concept. En effet, le concept d'arche, pour un être humain, englobe bien plus que la simple définition de connexions de segments : il comprend des notions d'architecture, mais aussi des références historiques, ainsi que tout un ensemble d'associations d'idées qui n'est d'ailleurs pas exactement le même pour chaque individu mais reste extrêmement riche.

En fait, dit Searle, la syntaxe (programme formel) est insuffisante pour produire la sémantique (le sens). L'argument qu'il emploie (Searle 1985; 1990) est celui de la *pièce chinoise* : supposons, dit-il, qu'on place dans une pièce un individu ne comprenant pas le chinois. Supposons également qu'on lui fournisse un ensemble d'instructions lui permettant, chaque fois qu'on lui donne un ensemble d'idéogrammes, de fournir une réponse qui soit correcte. Par exemple, à l'ensemble d'idéogrammes signifiant : « quelle est votre couleur préférée ? », les instructions lui disent d'écrire un autre ensemble d'idéogrammes qui signifie : « le bleu mais j'aime aussi le vert ». Notre individu ne sait pas ce que contiennent les idéogrammes. Les réponses qu'il fournit semblent prouver qu'il comprend le chinois alors qu'il se contente de manipuler un ensemble de symboles formels sans signification pour lui, et qu'il peut très bien, en fait, préférer le rouge. La mani-

---

4 Il y a des incorrigibles. Ainsi Patrick Winston dans (Winston 1984) écrit : « L'intelligence artificielle a pour but de rendre les ordinateurs plus intelligents afin qu'ils puissent remplir nombre de tâches comme la décision financière, la tenue ménagère de la maison... » D'autres, comme Alan Newell, estime que le Principe du Système de Symboles Physique est « certain ». Enfin, Friedkin est réputé avoir dit : « La prochaine génération d'ordinateurs sera tellement intelligente que nous aurons de la chance s'ils nous acceptent chez eux comme animaux domestiques. » La prudence est une chose qu'il est bien difficile d'apprendre.

pulation correcte des symboles (syntaxe) ne permet pas de supposer la compréhension correcte (sémantique)[5].

Le texte de Winston explique bien des désagréments qu'a rencontrés l'IA. On a trop tendance à croire que l'on peut réduire facilement un concept à un modèle de ce concept, alors que le modèle « oublie » systématiquement tout un ensemble de caractéristiques non fondamentales dans la définition mais indispensables pour assurer la connexité avec le reste du monde. C'est une des conséquences du réductionnisme de l'IA. Robert Escarpit[6] avait critiqué cette attitude (Escarpit 1981), directe conséquence du postulat mécaniste :

> « L'hypothèse mécaniste substitue aux individus des entités standardisées définies par quelques paramètres facilement quantifiables [...] Dans la mesure où un individu humain est, entre autres choses, un dispositif mécanique, l'information ainsi produite et traitée n'est pas fausse : elle est simplement incomplète et en particulier amputée de tout ce qui est pertinent au fait que l'individu humain est aussi un animal pensant et parlant. »

C'est probablement ce qui explique que l'IA a connu des succès sur ce que l'on appelle les *micro-mondes*, comme le monde du diagnostic médical ou de la synthèse chimique, mais a toujours échoué dès qu'il s'est agit d'étendre ces micro-mondes à un univers plus vaste. Les connaissances nécessaires au passage du micro-monde au monde réel sont absentes des modèles initiaux et bien difficiles à intégrer car très diffuses.

On ne peut qu'être frappé par ce que disait Wittgenstein (Wittgenstein 1961a) :

> « Considérons par exemple les processus que nous nommons « jeux ». J'entends les jeux de dames et d'échecs, de cartes, de balle, les compétitions sportives. Qu'est-ce qui leur est commun à tous ? — Ne dites pas : il faut que quelque chose leur soit commun, autrement ils ne se nommeraient pas « jeux », mais voyez d'abord si quelque chose leur est commun. — Car si vous le considérez, vous ne verrez sans doute pas ce qui leur serait commun à tous, mais vous verrez des analogies, des affinités, et vous en verrez toute une série. Comme je l'ai dit, ne *pensez pas* mais voyez ! Voyez, par exemple, les jeux sur damiers avec leurs multiples affinités. Puis passez aux jeux de cartes : ici vous trouverez beaucoup de correspondance avec la classe précédente, beaucoup de traits communs disparaissent, tandis que d'autres apparaissent. Si, dès lors, nous passons aux jeux de balle, il reste encore quelque chose de commun, mais beaucoup se perd. — Tous ces jeux sont-ils « *divertissants* » ? Comparez les échecs et la marelle. Ou bien y a-t-il en tous une façon de gagner et de perdre, ou une compétition des joueurs ? Songez aux patiences. Dans les jeux de balle on gagne et on perd ; mais quand un enfant lance une balle contre un mur et la rattrape, ce caractère se perd. Voyez quels rôles jouent l'adresse et la chance. Et combien différente est l'adresse aux échecs et l'adresse au tennis. Songez maintenant aux jeux de ronde : ici il y a l'élément du divertissement, mais combien d'autres caractéristiques ont disparu ! Et ainsi nous pouvons parcourir beaucoup d'autres groupes de jeux ; voir surgir et disparaître des analogies.
>
> Et tel sera le résultat de cette considération : nous voyons un réseau complexe d'analogies qui s'entrecroisent et s'enveloppent les unes les autres. Analogies d'ensemble comme de détail. »

---

5 On peut trouver une intéressante critique de cet argument dans (Churchland and Churchland 1990), Searle lui-même présente l'ensemble des objections qui lui ont été faites et les discute dans (Searle 1985).

6 La critique ne s'adressait pas à l'IA en particulier mais à l'informatique en général.

Ainsi, l'esprit humain semble plus fonctionner par analogies que par règles et maintient en permanence un flou dans ses représentations. Comme le reconnaissait Russell lui-même (Russell 1989b), cela est indispensable, pour la communication de tous les jours :

> « La question toute entière du sens des mots dans le langage ordinaire est pleine de difficultés et d'ambiguïtés [...] J'ai souvent entendu dire que c'était malheureux. C'est une erreur. Tout rapport serait impossible [...] parce que le sens que l'on attache à nos mots dépend de la nature des objets que l'on connaît directement, et que, puisque des gens différents ont une connaissance directe d'objets différents, ils ne seraient pas capables de parler les uns avec les autres. »

23.1.3 L'erreur de perspective

Cette critique s'adresse avant tout aux partisans de l'approche cognitive et se décompose en deux points.

Les cognitifs étudient le fonctionnement de l'esprit humain dans le but de dégager ses schémas de fonctionnement. Le problème majeur est qu'il n'est pas du tout évident que ces symptômes extérieurs, l'intuition que nous avons de notre raisonnement, ou la formalisation que nous en faisons, soient autre chose que la représentation de la technique apprise par le sujet, ne modélisant en aucun cas l'essence du fonctionnement de l'esprit humain qui devrait pourtant être le véritable but de l'étude.

Pour ceux qui douteraient de l'importance de la théorisation des problèmes et de l'influence de l'apprentissage, ils peuvent toujours imaginer un cogniticien de l'époque romaine tentant d'expliciter le fonctionnement de la multiplication (problème fort difficile, ce qui justifierait la mise en œuvre de techniques d'IA : essayez donc de multiplier CLXXXIV par LXXXIX sans recourir aux chiffres arabes...). Il est peu probable que le travail de notre psychologue-cogniticien ait eu le moindre intérêt une fois les chiffres arabes apparus. En revanche, il aurait peut-être permis de faire apparaître les failles du système de numération utilisé, si on l'avait considéré sous cet angle.

C'est là que se place la seconde critique. On peut penser que la progression scientifique ne s'est jamais réalisée en imitant l'esprit humain ou la nature, mais toujours en développant des théories qui permettaient à la technique de réaliser autrement ce que la nature ou l'esprit faisaient jusque là d'une autre façon.

L'approche cognitive présente donc un certain intérêt dans le domaine de la compréhension des erreurs et des raisonnements *dans un cadre culturel et technique donné*, ainsi que dans l'étude de l'influence de ce cadre sur l'individu. Elle peut ainsi aider à identifier, dans un processus de raisonnement, les parties routinières qu'il serait effectivement possible de faire exécuter à la machine plutôt qu'à l'homme. Il ne faudrait cependant pas prendre ces résultats à caractère purement local comme des énoncés généraux sur le fonctionnement de l'esprit humain, ni surtout comme une source indispensable de solutions à des problèmes.

Pour innover il est essentiel de pouvoir considérer un problème sous un angle original. Il ne faut pas s'étonner si la démonstration des théorèmes d'incomplétude et de non-consistance de l'arithmétique, ou la formulation de la théorie de la relativité furent l'œuvre de jeunes marginaux (Gödel et Einstein). L'école hilbertienne possédait certainement des esprits capables de concevoir la démonstration de Gödel, de même Poincaré

fut très proche de la relativité. Mais ils ne purent jamais se dégager complètement du carcan de leurs habitudes mentales. Il est bien plus important de se poser correctement un problème, que de tenter de le résoudre tel qu'il s'est toujours posé. C'est donc en cela que l'approche cognitive peut être conservatrice et sclérosante.

23.1.4 L'influence des résultats métamathématiques

Le théorème de non calculabilité vu au chapitre 5 stipule qu'aucune machine à architecture « classique » ne parviendra jamais à calculer l'ensemble des fonctions que peut imaginer un esprit humain. En conséquence, certains estiment que cela marque une frontière entre les hommes et les machines. Cette objection est faible dans la mesure où un être humain n'est pas non plus capable de les calculer, tout au plus peut-il définir l'ensemble de ces fonctions en extension, ce qui est très différent. Certains avancent le théorème d'incomplétude, pour prouver que les machines ne pourront jamais penser comme les êtres humains[7] ; là encore, il est difficile d'être pleinement convaincu. En fait, il semble peu probable que l'on puisse prouver qu'une IA est impossible.

23.1.5 L'argument quantique

Cet argument est dû au mathématicien Roger Penrose (Penrose 1992). Pour le résumer rapidement, R. Penrose pense que l'esprit humain utilise les propriétés quantiques de la nature, ce que ne fait aucun ordinateur. L'argument est brillamment présenté, mais laisse tout de même un peu sceptique dans l'état des connaissances actuel[8].

23.1.6 Critique théologique

Jacques Arsac a développé (Arsac 1987) une critique d'ordre théologique et humaniste :

> « Je serais donc gravement mutilé au plus profond de moi-même si je devais constater que mon intelligence est mécanique [...] Je serais déshumanisé s'il venait à être prouvé qu'une intelligence artificielle existe [...] Je ne crois pas, personnellement, que le débat puisse être tranché par l'expérience. La croyance en l'esprit est de l'ordre de la foi. C'est un acte libre, que l'on pose sur la base d'expériences personnelles, ou, comme je le crois, à l'appel d'une personne. S'il pouvait être déterminé par une expérience probante, il perdrait cette liberté et la foi cesserait du même coup. C'est une hypothèse qui me paraît tout à fait exclue. »

Le livre de Jacques Arsac est remarquable. Cependant, sa critique de l'IA présente l'inconvénient de placer *toute* l'IA sur un même pied, et en particulier les approches connexionniste, cognitive et pragmatiste qui s'opposent pourtant à l'intérieur de l'IA.

---

7 C'est la thèse défendue principalement par le philosophe J. R. Lucas.

8 Une autre critique a été faite par Turing dans l'article où il définit son célèbre test. Il prétend qu'une machine pourrait être distinguée d'un être humain car elle serait incapable de réaliser des expériences télépathiques, contrairement à un être humain. Il est très difficile de savoir si Turing était sérieux ou s'il s'agissait simplement d'un canular.

D'une part, il explique brillamment pourquoi on ne peut réduire un raisonnement humain à une suite de règles déductives, et ses citations de Pascal rejoignent le témoignage de nombre de chercheurs (Penrose, Schwartz, Poincaré...) sur la partie fulgurante de certains raisonnements, sur une intuition qu'il semble bien difficile de rattacher à un hypothétique processus calculatoire du raisonnement. Rien ne permet ni de montrer, ni même de supposer qu'il existe un niveau intermédiaire (cognitif) entre les niveaux phénoménologiques et les niveaux purement énergétiques (physico-chimiques) du cerveau. Scientifiquement, la position cognitive n'est qu'un postulat, et un postulat plus que discutable[9].

D'autre part, Jacques Arsac reprend également à son compte la position de Joseph Weizenbaum :

> « Joseph Weizenbaum s'insurge contre le fait que l'on puisse envisager sérieusement de confier à des ordinateurs la prise de décision dans des domaines où des personnes humaines sont en jeu. »

Il est bien difficile d'être systématiquement d'accord. Comment refuser, par exemple, de laisser un ordinateur contrôler un équipement médical de réanimation sous prétexte qu'une vie humaine est en jeu, alors que l'ordinateur est plus rapide, et, quoi que l'on en dise, plus fiable[10]. Dans nombre de domaines, il serait techniquement possible aujourd'hui de remplacer complètement l'homme par la machine ; mais des pressions culturelles et sociales s'y opposent (quel passager oserait monter dans un avion sans pilote ?), et la réaction de Jacques Arsac en est un exemple[11].

La critique de Jacques Arsac peut se reformuler de façon schématique en : « si une IA était possible, alors ce serait terrible pour un croyant, je me refuse à l'imaginer car ma foi n'y résisterait pas ».

23.1.7 Critique psychologique

Le psychologue Benny Shanon résume le modèle calculatoire[12] de l'IA en trois phrases[13] :

> « Le comportement des êtres humains est rendu possible par les connaissances contenues dans leur esprit.

---

9 En revanche, et sans bien distinguer les deux branches, Jacques Arsac s'en prend au neuro-réductionnisme, mais de façon nettement moins convaincante. Son attaque « théologique » contre J.P. Changeux et A. Lwoff semble scientifiquement plus douteuse. En effet, les neuro-sciences sont des sciences ; elles étudient des phénomènes physiques mesurables, et donnent des explications physico-chimiques de phénomènes qui se produisent au sein du cerveau humain. Elles n'en sont qu'à leur début, et ne peuvent aujourd'hui expliquer les processus les plus complexes, mais elles dégagent des mécanismes globaux (transmission des influx, niveaux de seuil, etc.) qui semblent applicables à l'ensemble du système.

10 Ainsi, Eric Hollnagel (Hollnagel 1991) estime à 80% la part des erreurs humaines dans les accidents. Notons que l'on rencontre de semblables chiffres dans les domaines du transport terrestre, aérien ou maritime. Et il est fort intéressant de constater que ce chiffre augmente régulièrement. L'article d'Eric Hollnagel discute les causes possibles de ce phénomène, et cite en particulier la fiabilité plus grande des systèmes automatiques.

11 On peut d'ailleurs se demander si la phrase de Jacques Arsac ne dépasse pas un peu sa pensée. Peut-être veut-il simplement dire que si une véritable intelligence artificielle existait, alors il n'accepterait pas qu'elle prit des décisions.

12 Nous traduisons l'anglais *computational* par *calculatoire*.

13 On pourrait la résumer en une seule, en citant la définition presque caricaturale que donne Jean-François Richard (Richard 1990) dans un traité de psychologie cognitive : « Ce qui caractérise les opérations mentales, c'est qu'elles construisent des représentations et qu'elles opèrent sur ces représentations. »

> – Ces représentations sont constituées par des structures symboliques bien formées.
> – Le comportement s'effectue au moyen d'une manipulation de symboles qui constituent ces représentations, c'est-à-dire au moyen d'opérations calculatoires. »

Les critiques que l'on peut adresser à ce modèle sont extrêmement nombreuses.

D'une part, il néglige l'aspect fondamental de la perception corporelle, dont l'importance était déjà soulignée par Husserl dès 1933 (Husserl 1976) :

> « Ainsi, la sensibilité, le fonctionnement égologique actif de la chair et des organes charnels, fait-elle essentiellement et fondamentalement partie de toute expérience des corps. »

Merleau-Ponty (Merleau-Ponty 1945) a développé cet argument, et Hubert Dreyfus l'a appliqué à l'IA très tôt (Dreyfus 1984). Il est intéressant de voir aujourd'hui un chercheur du MIT reprendre à son compte, sans le dire, certaines des critiques de Dreyfus : selon Rodney Brooks, un aspect fondamental est l'*embodiment*[14], ou nécessité de donner un corps à la machine en plus d'un cerveau[15]. Il décrit ainsi le but de son approche (Brooks 1991) :

> « Les robots sont situés dans le monde — Ils n'ont pas affaire à des descriptions abstraites, mais avec le ici et le maintenant du monde qui influencent directement le comportement du système.
>
> Les robots ont des corps et sont en rapport direct avec le monde — leurs actions sont une partie d'une dynamique avec le monde et ont ainsi un effet immédiat en retour sur leurs sensations.
>
> Les robots sont intelligents — mais la source de l'intelligence n'est pas limitée à un système de calcul. Elle vient également de la situation dans le monde, des transformations des signaux à travers les senseurs, et du couplage physique du robot avec le monde.
>
> L'intelligence du système émerge des interactions du système avec le monde et parfois des interactions entre ses composants — il est très difficile de dire quelle partie du système est responsable d'une action externe. »

Dans un ouvrage récent (Dreyfus and Dreyfus 1986), Hubert Dreyfus s'est également attaqué au postulat calculatoire de l'IA forte. Dreyfus commence par séparer, au niveau de l'être humain, plusieurs niveaux d'efficacité :

Le novice : il ne sait qu'appliquer un certain nombre de schémas fixes, de règles totalement déterministes indépendantes du contexte, donc facilement modélisables sur un ordinateur. Aux échecs par exemple, les schémas correspondent aux règles du jeu (déplacement des pièces, prises, mat) et à des assignations de valeur à chacune des pièces. Les règles correspondent à des séquences du type « si ... alors » comme par exemple : « *si* dans un échange de pièces, le total des points des pièces que tu perds est inférieur au total des points des pièces perdues par ton adversaire, *alors* réalise l'échange ». L'ordinateur étant plus rapide que le joueur humain dans ce type d'exercices, il écrasera aisément le débutant.

---

14 Que l'on pourrait traduire en français par *incarnation*.

15 Dans un livre récent (Varela *et al.* 1991), les auteurs désignent l'approche de Rodney Brooks sous le nom d'*enactive*.

Le pratiquant : à ce niveau, le joueur commence à acquérir une expérience du jeu qui lui permet de reconnaître des situations similaires et d'agir en conséquence en fonction d'expériences passées. Il peut ainsi reconnaître un roque affaibli ou un centre sous-développé, alors que ces notions ne sont pas immédiatement descriptibles en termes algorithmiques. Ce type d'expertise ne peut guère se communiquer qu'à l'aide d'exemples. Quiconque a pratiqué les échecs à un niveau un peu élevé sait qu'une bonne partie des livres d'échecs sérieux sont des collections d'exemples, illustrant un thème particulier, et qu'un des moyens principaux de la progression est l'étude des parties (les siennes et celles des champions) et non l'apprentissage de règles au moyen d'ouvrages[16].

L'individu compétent : à ce niveau de jeu, le joueur d'échecs commence à acquérir la notion de plan de jeu, ou *stratégie*. La mise au point d'un plan de jeu est difficile mais il commence à comprendre comment articuler son jeu pour réaliser ce plan et aussi quelles sont les positions où il faut appliquer un certain type de plan. La notion de plan semble très difficile à faire acquérir à un ordinateur[17].

La maîtrise : à ce niveau, le joueur devient capable d'élaguer automatiquement les coups inutiles ou les plans qui n'ont aucune chance d'aboutir. Cela se fait sans que le processus soit conscient.

L'expertise : un expert, par exemple un Grand Maître International d'échecs, ne considère plus l'échiquier comme un ensemble de pièces disposées sur une surface de jeu, mais comme un espace de possibilités dans lequel il se déplace automatiquement. À ce niveau de jeu, il peut réagir à une situation de façon automatique sans réflexion consciente et sans dépense de temps[18].

Il semble bien qu'un expert humain est presque toujours incapable d'exprimer son savoir sous forme de règles lorsqu'il est questionné par un cogniticien[19], et que le passage de son mode de raisonnement au système algorithmique doit toujours se faire par l'examen de cas particuliers que fournit l'expert, exemples que l'on tente de relier entre eux par un système de règles plaqué par le cogniticien. Ce mécanisme est finalement équivalent à celui qu'applique un bon joueur d'échecs pour apprendre à jouer à un novice, le passage à un niveau supérieur ne pouvant se faire que par la pratique intensive du jeu. Cette thèse permettrait également d'expliquer que le niveau de jeu d'un individu se dégrade lorsqu'il pratique moins, alors que sa connaissance du domaine et son aptitude à l'expliquer à un débutant ou un cogniticien peut se renforcer : c'est le cas de nombreux grands champions devenus pédagogues. Il n'y aurait donc pas oubli des règles associées au jeu, mais plutôt disparition de schémas cérébraux qui assurent la partie réellement experte du raisonnement.

---

16 Pour les gens intéressés par ce type de lecture, on peut recommander (Pachman 1980).

17 De plan stratégique s'entend. Il existe d'intéressantes tentatives pour faire acquérir des notions de plans tactiques aux ordinateurs.

18 Il faut savoir que un des premiers à avoir réalisé ce type d'observations est De Groot, et que ses travaux furent utilisés par l'équipe Simon, Newell, Shaw, en IA. Voir Dreyfus réutiliser ces résultats contre l'IA est particulièrement intéressant.

19 Un cogniticien, à ne pas confondre avec un cognitif (partisan de l'approche cognitive de l'IA) est une personne dont le but est « d'extraire » la connaissance d'un expert pour la communiquer sous une forme plus « programmable » à des informaticiens.

Le résultat final que pourrait obtenir un cogniticien programmant un système « expert » ne pourra donc jamais dépasser le niveau de novice avancé. Selon Dreyfus, l'idée de système *expert* est condamnée à la base, si l'on prend expert dans son sens littéral. La critique de Dreyfus vise avant tout l'approche qui veut reproduire le fonctionnement de l'esprit humain dans la machine. Elle ne s'applique pas aux mécanismes actuels de programmation du jeu, les techniques de la force brute, ou en général les techniques de parcours d'arbres que nous avons déjà examinées. En fait, elle s'oppose surtout à l'idée qu'on puisse réaliser un modèle calculatoire ou logique du raisonnement de type humain.

La critique de Dreyfus est en fait devenue celle de nombre de spécialistes aujourd'hui. Il est, par exemple, intéressant de constater l'évolution de Terry Winograd, qui fut un des pionniers de l'IA symbolique dans les années 70, avant de modifier radicalement son point de vue. En 1981, il écrivait : « la cognition n'est pas basée sur la manipulation de modèles mentaux ou de représentations du monde ». En 1986, il a largement développé ses critiques dans (Winograd and Flores 1986a).

Enfin, le modèle calculatoire prend souvent comme acquis tout un ensemble de fonctions « logiques » du cerveau. Combien de systèmes experts sont fondés sur une, ou des logiques ! Or, la vision logique du monde n'est qu'un garde-fou que nous utilisons pour vérifier la justesse de nos intuitions. Lorsque l'on résout un problème de mathématiques, domaine pourtant éminemment logique, on commence généralement par avoir une idée de la solution, une intuition du chemin à suivre, avant de vérifier techniquement la validité (et parfois la fausseté) de notre idée à l'aide des outils logiques que sont les techniques formelles de démonstration. Comme l'écrit le mathématicien Raymond Wilder :

> « [La démonstration] n'est rien de plus que la mise à l'épreuve des produits de notre intuition. »

Ce fait a été largement développé par Morris Kline (Kline 1989).

Or, les systèmes informatiques procèdent dans le sens inverse. Ils construisent une solution à partir de règles formelles, et s'efforcent de nous la présenter sous une forme qui nous soit intuitivement accessible : il y a là un renversement complet de la technique de « raisonnement ». Ne parlons donc pas des problèmes de la vie quotidienne, où la logique a bien peu de place. Il est bien difficile ensuite de croire à la possibilité d'une reproduction du raisonnement grâce à sa prétendue formalisation logique, alors que le raisonnement humain n'est pas logiquement formel.

Depuis quelques années, certains chercheurs s'occupant de psychologie cognitive critiquent eux-aussi cette approche « calculatoire » et algorithmique de l'intelligence humaine. Ils s'éloignent des modèles calculatoires parce qu'ils estiment que les activités mentales ne sont pas des opérations calculatoires mais bien plutôt des savoir-faire, évoluant en permanence et dépendant du contexte (Shanon (Shanon 1991)). Bien entendu, l'école Gestaltiste s'est toujours refusée à accepter le modèle calculatoire cognitif[20] ; les Gestaltistes se sentent proches de la vision connexionniste. Ainsi, Irvin Rock et Stephen Palmer écrivent (Rock and Palmer 1991) :

---

20 Signalons cependant que la psychologie cognitive évolue sous la pression des modèles connexionnistes. Il suffit de lire (Tiberghien 1988) pour s'en apercevoir. Mais il n'est peut-être pas encore bien clair, aux yeux des psychologues, que l'adoption d'un modèle connexionniste signifie la fin de la psychologie cognitive. Car dans ce type de modèle, il n'y a plus rien à formaliser, et plus rien à interpréter ; la connaissance est éclatée sous forme de coefficients, de poids analogiques sans signification, sans aucun « sens ». Le réseau n'est qu'une boîte noire, et sa description ne concerne que les mathématiciens ou les biologistes.

> « Le remarquable renouveau d'intérêt concernant les modèles de réseaux de neurones atteste la viabilité des idées de la Gestalt. [...] L'idée plus générale que le cerveau est un système dynamique convergeant vers un état d'équilibre grâce à une fonction énergétique pourrait se révéler. »

23.1.8 Critique biologique

Parallèlement, les chercheurs en neuro-biologie ont une position très proche de celle des Gestaltistes. Pour eux, le fonctionnement du cerveau humain provient de deux choses : d'une part, une structure « pré-câblée » importante, résultat de l'évolution biologique, qui permet un accès efficace aux perceptions de l'environnement[21] ; d'autre part, des mécanismes chimiques ou électriques opérant à l'intérieur de cette structure physique, probablement proches de ce que les Gestaltistes défendent. On peut citer un article récent reprenant cette double vision des choses (Mahowald and Mead 1991) :

> « L'efficacité de la rétine surpasse celle du plus puissant des super-ordinateurs. Pourtant chaque neurone rétinien est environ un million de fois plus lent. [...] Manifestement, les systèmes de calcul biologiques diffèrent beaucoup de leurs homologues informatiques. [...] Les circuits classiques de traitement de l'image diffèrent radicalement des circuits neuronaux de la rétine humaine. Ils sont constitués d'une matrice photo-sensible [...] et d'un calculateur numérique puissant qui extrait ensuite les caractéristiques géométriques de ces données numériques. La rétine au contraire comporte cinq couches de cellules nerveuses au travers de laquelle l'information circule à la fois verticalement et horizontalement. Dans la rétine, le traitement des photons et le traitement des informations lumineuses sont indissociables. Nous pensons que l'architecture rétinienne est déterminante dans la formation des images visuelles[22]. »

De même, une étude a été menée sur la simulation de lésions cérébrales (Hinton *et al.* 1993) :

> « Quand ils lisent, les patients victimes de lésions cérébrales font des erreurs étranges. Quand un réseau de neurones est endommagé après qu'il ait appris à lire, il reproduit le même type d'erreurs. »

Il n'y a point ici de manipulation de symboles[23], mais bien une approche *globale* de la perception, basée sur des phénomènes chimiques et physiques.

---

21 Des résultats récents semblent clairement aller dans ce sens (Elmas 1985).

22 En revanche, dans un traité récent de psychologie cognitive, le cerveau humain était modélisé par un ordinateur numérique classique, avec unité centrale de calcul, mémoire, mémoire de masse, unités d'entrée-sortie (!!!)... Que l'on puisse réaliser ce type d'analogies n'est pas tellement surprenant : von Neumann a conçu l'architecture des calculateurs (cf. (Siri 1993)) en se basant, avec beaucoup de prudence, sur les connaissances de la neurobiologie des années 40 : « Le neurone individuel peut-être—tout au moins dans des situations particulières—un mécanisme bien plus compliqué que ne peut l'exprimer une description dogmatique en termes de stimulus et de réponse, selon les structures simples des opérations logiques élémentaires. » ((von Neumann 1992)).

Voir la psychologie cognitive des années 90 chercher des modèles du fonctionnement cérébral dans une architecture de calculateurs vieille de 50 ans est un retournement de perspective que l'on pourrait trouver comique, s'il ne manquait d'inquiéter quant au sérieux de cette discipline, trop souvent à la recherche de justifications pseudo-scientifiques.

23 Nous prenons *symbole* dans l'acception que lui donne le principe du système de symboles physiques de Newell, Simon et Shaw, tel que nous l'avons présenté dans le chapitre 1.

De nombreux auteurs s'appuient sur les résultats de la biologie moderne pour montrer que le fonctionnement du cerveau n'a rien du fonctionnement d'un système symbolique. Rodney Brooks, dans son excellent article (Brooks 1991) fait aussi un bilan des arguments que la biologie et la psychiatrie apportent contre le modèle symbolique de l'IA. La conclusion est claire : le cerveau ne peut se résumer à un système symbolique. Comme l'écrit également Teuvo Kohonen (Kohonen 1988a) :

> « Il n'y a ni instructions machines ni codes de contrôles au niveau du fonctionnement cérébral. [...] Les circuits cérébraux n'utilisent pas de méthodes récursives de calcul, et ne sont donc pas des algorithmes. [...] Même au plus haut niveau, la nature de l'information traitée est différente dans un ordinateur et dans le cerveau. »

Ces critiques ont amené le développement des techniques connexionnistes (réseaux de neurones), ainsi que les techniques cybernétiques modernes dont Brooks est un défenseur. Certains biologistes semblent complètement convaincus de l'efficacité de cette approche (Alkon 1989) :

> « Nous avons déterminé des règles utiles à la conception des « réseaux de neurones formels », des systèmes informatiques de stockage de l'information et de reconnaissance des formes. Le réseau artificiel que nous avons construit en copiant le cerveau fonctionne bien ; en retour, les formules mathématiques qui décrivent le fonctionnement de ce réseau ont révélé certains mécanismes de la mémoire. »

23.1.9 Optimalité de l'évolution

Une des questions que l'on peut raisonnablement se poser est de savoir si l'évolution biologique naturelle n'a pas déjà fait « au mieux » au long des siècles d'évolution du cerveau humain. On ne parviendrait alors pas à construire un modèle moins complexe et réalisant les mêmes fonctions que le cerveau. On peut citer comme exemple original de cette parcimonie de l'évolution, les intéressants travaux de Thomas Ray. Il a tenté de simuler sur ordinateur, non pas une intelligence, mais une forme de synthèse de la vie. La description de l'ensemble du système serait trop longue, mais le principe est le suivant : on place dans une « soupe informatique » un programme élémentaire dont le but est simplement de se dupliquer. On introduit des facteurs de mutations, et on laisse tourner la machine. On constate plusieurs phénomènes, dont l'apparition de programmes parasites utilisant le code d'autres programmes pour survivre. Et on constate aussi l'efficacité du code des organismes évoluant. Certains d'entre eux arrivent à être même plus efficace que le programme original (Ray 1991) :

> « Experiments have shown that freely evolving digital organism can optimize their algorithm by a large factor in a few hours of real time[24]. »

Le problème se situe à un semblable niveau : peut-on reconstruire plus simplement ce que la nature a fait au cours de millions d'années d'évolution ? Rien n'est moins sûr.

---

24 « Des expériences ont montré que des organismes digitaux évoluant librement peuvent optimiser leur algorithme d'un facteur important en quelques heures. ».

23.2 Une IA est-elle impossible ?

Nous avons vu dans la section 23.1 nombre de critiques opposées à la possibilité d'une IA, la plus radicale[25] étant celle de John Searle. Pourtant, aucun de ces arguments ne semble catégorique. Est-il possible de prouver scientifiquement qu'une IA est impossible ?

Il est bon de préciser les hypothèses de travail :

- Le cerveau est le siège de la pensée et peut être totalement décrit par des mécanismes chimiques et physiques. Cette opinion est acceptée par la plupart des gens, à l'exception de ceux défendant l'origine divine de l'intelligence. Elle a été très clairement défendue par John Searle dans (Searle 1985).
- Il n'y a aucune impossibilité théorique quant à la reconstruction d'un cerveau humain à l'aide de composants électroniques. Chaque neurone pourrait être remplacé par un équivalent électronique remplissant les mêmes fonctions. Cette hypothèse est déjà plus difficile à accepter car elle apparaît comme parfaitement irréaliste en rapport à la technologie actuelle et à notre connaissance encore incomplète du cerveau.
- Ayant ainsi reconstruit le cerveau, rien n'empêche de reconstruire le corps pour répondre à l'objection d'Hubert Dreyfus quant à la nécessité des perceptions et de l'action pour qu'il y ait intelligence.

L'IA ainsi réalisée est un système entièrement caractérisé par les poids des coefficients des liaisons et les mécanismes faisant évoluer ses poids pour chacun des neurones. En fait, même Dreyfus et Searle seraient en mesure d'accorder le raisonnement ci-dessus ; ils appelleraient sans doute cet « objet » un artefact et non une IA, mais peu importe. Quant aux partisans de l'IA, ils n'ont aucune raison de contester ce qui précède.

Ainsi, Dennett et Hofstadter acceptent ce raisonnement, mais ils le complètent par la notion de *niveau de description*. Pour eux, s'il est sans doute exact que le cerveau fonctionne bien ainsi à un niveau élémentaire, il faut se placer à un niveau supérieur de description qui permettra d'avoir une vision symbolique globale du fonctionnement. *Ils résument ainsi de la meilleure façon la véritable thèse de l'IA forte* : celle-ci prétend qu'il existe un modèle symbolique « plus simple », permettant d'obtenir les mêmes résultats que ce modèle naïf qui peut être considéré comme une simple copie de l'original[26].

C'est ce qui autorise les tenants de cette approche à dire que l'on peut faire directement de l'apprentissage de concepts, et parler d'états mentaux de « hauts niveaux » qui se situent entre la perception immédiate des phénomènes et le niveau élémentaire physique et chimique. C'est ce qui justifie, en fait, l'ensemble des recherches en IA symbolique forte et en psychologie cognitive : si le modèle est réductible, alors les recherches aboutiront. Sinon, elles échoueront toujours, car il sera impossible de décrire les états mentaux autrement qu'en les recréant.

C'est ici qu'il est bon de rappeler la définition de la complexité de Chaitin-Kolmogorov : la complexité d'un système (voir chapitre 11) au sens de Chaitin est précisément la « compressibilité » des nombres qui le caractérisent. Ainsi, les 100 premiers nombres pairs ont une complexité très inférieure à 100, car on peut les décrire en une formule beaucoup plus courte. L'hypothèse forte se ramène donc formellement à ceci :

---

25 Hors la critique théologique qui règle définitivement le problème.

26 Voir Bernd-Olaf Kuppers (Kuppers 1989).

la complexité de Chaitin du modèle cérébral est inférieure (et de beaucoup) à la taille du modèle naïf décrit.

Or, cette hypothèse présente la caractéristique suivante : si elle est fausse, cela est non-démontrable. C'est une conséquence directe du théorème de Chaitin qui énonce qu'il est impossible de prouver que le modèle a atteint la taille minimale. On peut résumer cette courte discussion en une phrase lapidaire : *la proposition « une IA forte peut exister » est soit vraie soit indémontrable.*

Nous pensons donc qu'il est plus que probable qu'il n'existera jamais aucun argument définitif permettant de *prouver* que l'on ne peut réaliser une IA suivant la thèse symbolique forte.

23.3 Qu'en penser ?

23.3.1 Différentes attitudes vis-à-vis de l'IA

Essayons de classer, de la plus pessimiste à la plus optimiste, les différentes attitudes sur l'IA :

1. Il est impossible qu'une IA existe car l'intelligence a été donnée à l'homme par Dieu (Jacques Arsac).
2. L'intelligence repose sur des mécanismes biologiques, et il faudrait complètement reconstruire (au sens moléculaire) le cerveau et le corps pour la simuler (John Searle, Hubert Dreyfus).
3. L'intelligence est une propriété d'un système qui doit *nécessairement* être associée à un corps (embodiment theory). D'autre part, l'intelligence est une propriété *émergente*, et il faut donc utiliser une technique de bas en haut (bottom-up) et non une technique de haut en bas (top-down) pour y parvenir. On parviendra peut-être à simuler une intelligence par la construction de systèmes complets mais très simples (le « robot-fourmi ») que l'on rendra lentement complexes (Rodney Brooks).
4. L'intelligence repose sur des mécanismes de type connexionnistes et parallèles, et il faut développer ce type de machine et d'architecture pour simuler un cerveau « artificiel » (Paul et Patricia Churchland).
5. Les machines actuelles ne sont certainement pas adaptées pour réaliser une véritable intelligence, mais on pourra probablement en construire de meilleures, et on peut penser que l'on pourra utiliser des méthodes de type symbolique (Dennett, Hofstadter).
6. Les machines actuelles peuvent permettre de réaliser une véritable IA. Simplement, l'état de la recherche n'est pas suffisamment avancé. Il faut, par exemple, développer les techniques de représentation de connaissances et d'apprentissage symbolique (Schank).
7. Il existe déjà des machines qui sont intelligentes ou qui sont proches de l'être (Newell, Simon, McCarthy).

23.3.2 Un avenir pour l'IA

Résumons en quelques lignes ce que sera probablement l'avenir de l'IA :

- Les ordinateurs actuels (basés sur une architecture type machine de Von Neumann) sont des objets formidables pour faire des *calculs* et stocker des données numériques. L'accroissement de la vitesse et de la mémoire de ces machines va certainement se poursuivre, mais il semble peu raisonnable d'imaginer qu'une programmation symbolique *a priori* puisse un jour aboutir à une manifestation générale de type « intelligent ». Il faut, pour l'instant, les utiliser pour ce qu'elles savent bien faire, tenter d'améliorer les méthodes mathématiques permettant de les utiliser au mieux, et cesser de disserter sur l'intérêt d'un *modèle cognitif de l'homme* pour la programmation de ces machines. Comme le dit très justement Jonathan Schaeffer[27] :

> « [Il y a] une théorie qui dit que la méthode la plus simple pour construire une machine intelligente est de modéliser l'être humain. Personnellement, je trouve qu'il s'agit d'une approche peu perspicace. Je considère l'esprit humain comme une machine, comme je considère mon ordinateur comme une machine. Ces deux machines ont leurs forces et leurs faiblesses. Mon cerveau est très fort pour la pratique du langage naturel et la reconnaissance de formes. Mon ordinateur est très bon pour répéter des tâches et faire des calculs. En revanche, mon cerveau ne peut s'occuper que d'un petit nombre de tâches répétitives et j'ai de sérieux problèmes à additionner un million de nombres en une seconde. Mon ordinateur n'est pas très doué pour le langage naturel ou la reconnaissance de formes.
>
> Étant donné que chaque machine a des capacités différentes, il faut essayer d'exploiter (à mon sens) les forces des machines et non leurs faiblesses. Il faut prendre avantage de la capacité des machines à exécuter des calculs mathématiques rapidement. Donc, la force brute est la technique normale.
>
> Malgré tous nos efforts pour inclure de la connaissance dans CHINOOK, le programme est toujours à la base un système de recherche à force brute. Il y a bien sûr de la connaissance pour aider à guider la recherche. Mais cela est secondaire par rapport aux bénéfices que nous obtenons par la vitesse de calcul. »

Ainsi :

1. Face à un problème formellement spécifiable et d'une complexité raisonnable[28], l'ordinateur peut être rendu plus efficace que l'être humain. Il n'y a là aucun scandale, aucun paradoxe et aucune inquiétude à avoir. Au contraire, l'ordinateur permet ainsi à l'homme de se dégager de tâches répétitives souvent sans grand intérêt.
2. Face à des problèmes mal spécifiables, faisant intervenir des connaissances vastes et mal définies, l'homme sera longtemps encore plus efficace que les

---

27 Communication privée. Rappelons que Jonathan Schaeffer est l'auteur, avec son équipe, du programme de dames américaines CHINOOK qui s'est qualifié pour la finale du championnat du monde en terminant second de l'Open des États-Unis derrière le champion du monde en titre (chapitre 15).

28 La frontière entre ce qui est raisonnablement complexe et ce qui ne l'est pas est floue. Elle évoluera en fonction de l'amélioration des algorithmes et de la puissance des machines.

ordinateurs à architecture classique. Les échecs de l'IA cognitive sur les trente dernières années sont bien là pour montrer qu'une approche symbolique *a priori* ne permet pas de les traiter.

- La notion de *modèle cognitif symbolique* de haut niveau est peu vraisemblable. Le fonctionnement du cerveau humain n'est symbolique qu'à un niveau superficiel, si jamais il l'est, et le cerveau humain n'est pas réductible à un système symbolique construit *a priori* et dont la complexité serait très nettement inférieure à la description extensive du système.
- Il est possible que l'on puisse, par exemple, simuler le cerveau en s'appuyant sur des mécanismes autres que strictement biologiques ou chimiques, en modélisant parfaitement le fonctionnement de chaque neurone, et en espérant représenter parfaitement tous les types d'interactions possibles. Cependant, si l'on parvient à réaliser de semblables machines (ce qui est loin d'être sûr) :
  1. Elles seront probablement très différentes de ce que nous pouvons imaginer aujourd'hui.
  2. Elles ne seront pas faites demain. Nos connaissances à la fois technologiques (sur les composants, les possibilités de connexions) et biologiques (sur le cerveau) sont infiniment trop peu avancées.
  3. L'IA sera peut-être alors une propriété émergente des systèmes et non pas une propriété programmée *a priori*.

23.4 L'IA : une grande illusion ?

Comment se peut-il que tant de chercheurs continuent à traiter l'IA comme une religion, une foi qu'il faut embrasser, et non comme une discipline scientifique, alors qu'à l'opposé, d'autres chercheurs tout aussi compétents et reconnus crient au sacrilège et appellent à la révolte de l'homme ? Il y a des raisons historiques et sociales à cette attitude, et nous allons essayer de les dégager.

23.4.1 Rapide analyse historique

En fait, les origines sociales et historiques du problème sont loin d'être simples. Il serait trop long de refaire ici un historique complet de l'évolution de la pensée humaine à travers l'histoire et de voir en détail l'influence que la philosophie a pu avoir sur le développement des sciences, mais aussi l'influence que la science a eu sur la philosophie. Hubert Dreyfus traite largement le problème dans son ouvrage. On peut également se reporter aux travaux d'Alexandre Koyré (Koyré 1966a; 1966b), ainsi qu'au livre de Kuhn (Kuhn 1983), ou aux derniers travaux de Husserl (Husserl 1976) : on aura, à travers ces ouvrages, une vision complémentaire de l'évolution scientifique et philosophique « moderne ». John Haugeland (Haugeland 1989), dans une étude des fondements de l'IA, a également fait une présentation du cheminement de la pensée occidentale d'Aristote à l'apparition de l'IA, à travers l'empirisme anglo-saxon. L'excellent livre de Jean Largeault (Largeault 1988) dresse également un panorama très complet de l'évolution des concepts fondamentaux des sciences.

Disons simplement que depuis la fin du Moyen Âge, la science occidentale est sortie de la conception aristotélicienne du monde pour entrer dans ce qu'Alexandre Koyré appelle la conception néo-platonicienne[29] :

> « Le platonisme et le néo-platonisme ont toujours eu tendance à traiter par les mathématiques les phénomènes naturels. »

En un mot, la science moderne va donner la prééminence à l'interprétation mathématique contre l'interprétation aristotélicienne des causes[30]. Il restera bien sûr à débarrasser le platonisme de sa partie spiritualiste pour fonder la philosophie et la science occidentale, ce que feront Descartes et Galilée[31].

La conception platonicienne est indissociable de l'Idéalisme, théorie qui marquera durablement, sous différentes formes, la philosophie occidentale ; on admet l'existence de concepts purs, d'idéalités qu'il est possible d'atteindre (au travers, certes, d'une suite infinie d'approximations) par la pensée pure[32] (Husserl 1976) :

---

29 En fait, il s'agit aussi et surtout d'une conception pythagoricienne. Pour une discussion plus générale de l'influence des diverses écoles pré-socratiques, on peut lire (Dumont 1991). Il ne faudrait cependant pas croire que les conceptions pythagoriciennes et occidentales du nombre sont aussi proches qu'ont pu le dire et le penser nombre d'auteurs, comme le montre brillamment Spengler (Spengler 1948) : « Il n'y a pas une, il n'y a que des mathématiques [...] L'aphorisme de Pythagore selon lequel le nombre représente la nature de toutes les choses *accessibles aux sens*, est resté le plus précieux de la mathématique antique. Par lui, le nombre est défini comme une mesure, et cette définition renferme le sentiment cosmique tout entier d'une âme orientée vers l'actuel et le tangent. [...] De par son sentiment cosmique tout entier, la pensée antique mûre ne pouvait voir dans la mathématique que la doctrine des rapports de grandeur de mesure et de figure entre corps incarnés. Partant de ce sentiment, si Pythagore a prononcé la formule décisive, c'est que le nombre était pour lui précisément un symbole *optique*, non une forme en général ou un rapport abstrait. Toute l'antiquité sans exception concevait les nombres comme des unités de mesure, des grandeurs, des lignes ou des plans. Une autre espèce d'étendue lui est inconcevable [...] L'antiquité ne connaît donc que des *nombres naturels* dont le rôle est tout à fait effacé parmi les espèces numériques nombreuses et extrêmement abstraites de la mathématique occidentale [...] Euclide disait que les nombres incommensurables « ne se comportent pas comme les nombres ». Il y a en effet dans le concept réalisé du nombre irrationnel la séparation complète entre le concept de *nombre* et celui de *grandeur*, car un nombre, comme π par exemple, ne pourra jamais être ni délimité ni représenté exactement par une ligne [...] Un mythe étrange de la Grèce tardive veut que le savant qui a divulgué le premier le secret de l'irrationnel périt dans un naufrage, « parce qu'il fallait que l'ineffable et l'invisible restassent à tout jamais cachés. » Quiconque sent l'angoisse qui est à la base de ce mythe a compris aussi et le sens du nombre antique, *mesure* opposée à l'incommensurable et le haut éthos religieux de son emploi restreint. L'acte décisif de Descartes fut d'avoir conçu une nouvelle idée du nombre, consistant principalement à débarrasser la géométrie de la manipulation optique de la construction, soit de l'espace mesuré et mesurable. Au lieu de l'élément sensible de la ligne et du plan concret apparaît l'élément abstrait spatial, le point, désormais caractérisé comme un groupe de nombres purs conjugués [...] Il supprimait ainsi la géométrie en général qui ne mènera qu'une existence apparente que couvrent des réminiscences antiques. »

30 Il est intéressant de constater à quel point la conception mathématiste que nous admettons aujourd'hui comme allant de soi, est en fait peu intuitive. Ainsi, un des phénomènes les plus étudiés par les différentes écoles a été le problème du mouvement des corps. Le Moyen Âge a développé la théorie dite de l'*impetus*, qui disait qu'un corps lancé avait acquis un *impetus* comme un corps chauffé a acquis de la chaleur, et que cet *impetus* s'épuisait durant le mouvement comme un corps se refroidit. Cette théorie me paraît intuitivement bien plus facile à accepter que la théorie moderne qui ne fait aucune différence entre un corps en mouvement et un corps au repos.

31 Il est difficile de savoir la raison exacte qui poussera l'église catholique à s'opposer aux idées galiléennes. On peut penser que, plus que le contenu technique de l'œuvre de Galilée, c'est l'influence platonicienne qui est combattue. En effet, la conception catholique est que la Grâce qui touche l'homme est une Re-naissance au sens propre : Dieu apporte la connaissance. Au contraire, la conception platonicienne est innéiste : il suffit à l'homme de se resouvenir de ce qu'il sait déjà ; l'âme peut exister sans qu'il soit réellement besoin de Dieu (Kierkegaard 1967). L'opposition entre théologiens néo-platonicien et aristotélicien aura toujours existé à l'intérieur de l'église. On peut lire en particulier (Gilson 1947), qui montre d'ailleurs que le néo-platonicisme pose d'autres problèmes, dont l'immortalité des âmes et la résurrection des corps.

32 La pensée occidentale modifiera le concept platonicien en y incluant la notion d'*infini achevé*.

> « La conception d'une telle *Idée d'une totalité d'être rationnelle infinie, systématiquement dominée par une science rationnelle,* voilà la nouveauté inouïe. C'est la conception d'un monde infini (ici d'un monde des idéalités), tel que les objets de ce monde ne soient pas accessibles individuellement, incomplètement et presque par hasard à notre connaissance, mais soient atteints par une méthode rationnelle systématique unifiée, qui finalement, dans une progression infinie, atteint *tout* objet dans la plénitude de son être-en-soi. »

Ainsi le postulat fondamental de toute la connaissance occidentale est qu'il est *possible* de se rapprocher aussi près qu'on le désire de la Vérité du monde : le monde est la limite de nos approximations scientifiques et philosophiques.

Il faut remarquer la fécondité et l'efficacité de ce modèle qui, sur le plan scientifique, aura résisté plus de trois cents ans et aura provoqué une formidable évolution culturelle. Sur le plan philosophique, le modèle connaîtra plus rapidement des dissensions internes. En fait, la philosophie cartésienne, dès ses débuts, contiendra en germe les deux pôles de la philosophie occidentale : l'empirisme des philosophes anglais et le courant idéaliste allemand. Et les modèles philosophiques occidentaux ne parvinrent jamais à converger :

> « La philosophie universelle, dans laquelle ces problèmes étaient liés aux sciences de fait, prit la forme de systèmes philosophiques impressionnants, mais qui malheureusement manquaient d'unité entre eux et même qui divergeaient.[...] Vint alors une longue période de combats pénibles qui s'étend de Hume et de Kant jusqu'à nos jours, en vue d'obtenir la claire compréhension de cet échec séculaire. [...] La philosophie devint à elle-même un problème, et d'abord ce qui est bien compréhensible, sous la forme de la possibilité de la métaphysique. »

Or, comme le fait remarquer Husserl, cela devait entraîner une crise sur les fondements de la rationalité occidentale :

> « C'est là une crise qui n'entame pas ce qui est scientifique dans le sens artisanal du terme et qui pourtant ébranle de fond en comble le sens de la vérité de la science dans son ensemble.[...] Du même coup tombe également la foi en une raison « absolue », d'où le monde tire son sens, la foi en un sens de l'histoire, en un sens de l'humanité [...]
>
> Dans la détresse de notre vie, — c'est ce que nous entendons partout — cette science n'a rien à nous dire. Les questions qu'elle exclut par principe sont précisément les questions qui sont les plus brûlantes à notre époque malheureuse pour une humanité abandonnée aux bouleversements du destin : ce sont les questions qui portent sur le sens ou sur l'absence de sens de toute cette existence humaine. »

Le divorce de la philosophie et de la science sera ressenti par nombre de mathématiciens qui s'intéresseront à la philosophie dans le but de construire une nouvelle philosophie formelle plus solide ; Bertrand Russell sera un des plus fameux. Il sera un des défenseurs de la thèse de l'atomisme logique[33], dont le premier Wittgenstein (Wittgenstein 1961b) sera un proche parent. Kurt Gödel consacrera également les trente dernières années de sa vie à tenter de construire une philosophie définie ainsi (d'après Hao Wang (Wang 1990)) :

> « Gödel veut dire que la relation entre la logique et les concepts (ou le concept de concept) est entièrement analogue à la relation existant entre les mathé-

33 Pour plus de précisions on peut lire (Russell 1961; 1989a).

> matiques et les ensembles. Il souhaite également dire que la relation entre la logique et la métaphysique est complètement analogue à la relation existant entre les mathématiques et les ensembles. »

La philosophie de Gödel semble, à bien lire le livre de Hao Wang, proche de celle de Leibnitz. Comme Leibnitz, il se réfère à un principe monadologique, faisant de chaque individu une image du monde dans son intégralité. Les logiciens[34] se sont en fait appropriés une partie de la philosophie moderne : Quine, Carnap, Ramsey, Berneys et Hao Wang lui-même sont aussi des « philosophes ». Mais la grande ambition de créer un système unifié s'est éloignée.

En fait, les sciences occidentales ont elles-mêmes perdu une partie de leur crédibilité. Les mathématiques tout d'abord, à travers les crises qu'elles ont traversées : la crise déclenchée par Russell, le théorème d'incomplétude de Gödel et ses interprétations[35] ont amené les mathématiciens eux-mêmes à douter de la valeur universelle de leur science. Il en va de même en physique où la mécanique quantique a sérieusement ébranlé les fondements de la science moderne[36]. Le principe même de la pensée occidentale, qui voulait que les concepts idéaux puissent être, au moins théoriquement, parfaitement atteints par approximations successives, est naïvement battu en brèche par le principe d'incertitude ; quant à l'idée d'une logique pure, elle se heurte à l'incapacité que nous avons de construire un système formel mathématique un peu riche, dont on puisse prouver qu'il soit complet. La science semble brusquement se révéler incapable de saisir le monde à travers une raison objective. Elle connaît une évidente crise de foi.

Pourtant, ce n'est pas tellement la mécanique quantique qui est paradoxale, que les interprétations et les analogies macroscopiques que l'on en fait[37]. De même, le théorème de Gödel n'est pas un paradoxe. Tous ces résultats ne portent pas atteinte au réalisme scientifique, mais posent plutôt une borne à la connaissance scientifique et à la formalisation. S'il y a crise de la science, elle porte plus sur ses limites que sur ses fondements[38].

Cela peut sembler étonnant quand on constate les progrès technologiques accomplis dans les quarante dernières années, mais il faut pourtant reconnaître, à l'opposé, l'absence de découverte scientifique fondamentale. En fait, comme le fait justement remarquer Alexandre Koyré, le progrès technologique n'a que peu de rapport avec le progrès scientifique :

> « Des merveilles telles que l'arc gothique, les vitraux, le foliot ou la fusée des horloges et des montres à la fin du Moyen Âge, n'ont pas été les résultats des progrès des théories scientifiques correspondantes et n'ont pas suscité celles-ci non plus. »

Il semble clair que l'Occident se trouve actuellement dans une phase de crise intellectuelle. Jacques Arsac parle, quant à lui, d'une « crise du sens » et le constat semble général : pour certains « Voici venir le temps du monde fini » (Albert Jacquard), pour d'autres,

---

34 Il n'est pas très étonnant que ce soient les logiciens qui se soient le plus sérieusement intéressés au problème : « L'objet de la logique est la forme de la science » (Lachelier 1990), et cette forme est sérieusement remise en question.

35 Nous avons largement développé ce sujet dans le chapitre 5.

36 Rappelons pour mémoire les deux références bibliographiques (d'Espagnat 1989;).

37 On s'aperçoit bien de ce phénomène lorsque l'on pratique un peu la théorie quantique.

38 Sur le problème de la crise du réalisme en sciences, on peut lire l'article de Jean-Paul Delahaye (Delahaye 1991).

nous vivons dans « une douce certitude du pire », d'autres enfin rappellent Dieu au secours (Jean Guitton, Igor et Grichka Bogdanoff), ou tentent de conjurer « La peur du vide » (Olivier Mongin).

La philosophie n'est pas parvenue à prouver ce qu'elle croyait, la science ne parvient plus à croire ce qu'elle prouve[39].

23.4.2 L'IA, en dernier recours

Dans cette crise générale de la société et de la science contemporaines, chacun tente de se raccrocher à une bouée de sauvetage. L'intelligence artificielle cognitive fait partie de ces grands phénomènes de mode qui, sous couvert d'une « nouvelle science » voulant replacer l'homme « au centre de l'univers », ouvrent la voie à des divagations fort discutables. Les modèles cognitifs sont devenus la dernière bouée d'un réductionnisme outrancier, une survivance caricaturale de l'atomisme logique de Russell. Le paradoxe est que l'on tente de les défendre au nom d'un certain humanisme alors qu'ils représentent ce que les sciences sociales ont de pire : la tentation de ressembler à une axiomatique, à une science formelle, alors qu'elles n'en sont pas. Comme le dit Claude Lévy-Strauss :

> « Les « sciences humaines » ne sont des sciences que par une flatteuse imposture. Elles se heurtent à une limite infranchissable, car les réalités qu'elles aspirent à connaître sont du même ordre de complexité que les moyens intellectuels qu'elles mettent en œuvre. De ce fait, elles sont et seront toujours incapables de maîtriser leur objet. »

Il ne faut pourtant pas, bien au contraire, rejeter les sciences sociales ou humaines : elles participent au *corpus* de connaissances *indispensables* au savoir humain et à son avancée. La psychologie est une discipline fondamentale à condition de la maintenir dans le cadre qui est le sien[40]. Il faut également réhabiliter l'enseignement de la philosophie et sa pratique, car seule une interrogation sur les bases des sciences et du savoir permet d'appréhender les forces et les faiblesses de leurs fondements[41].

Bien qu'ils soient les défenseurs de thèses philosophiques fort anciennes, de nombreux partisans du modèle cognitif prétendent être les chevaliers d'un nouveau paradigme révolutionnaire. Ils s'estiment victimes de « scientifiques dépassés » ou de « philosophes archaïques » rejetant par conservatisme leurs nouvelles idées. Certes, le rejet des dogmes et des paradigmes acceptés fait partie de la nécessaire évolution de la science. Encore

---

39 Il reste cependant une partie de la science qui semble encore connaître un développement sans état d'âme : la biologie. Il est particulièrement intéressant de lire (Changeux and Connes 1989) pour constater que les biologistes doutent moins de leur science, et de la science en général, que les mathématiciens. Un état de fait probablement lié à la jeunesse de leur discipline.

40 « Pourquoi toute psychologie, dès qu'elle cesse d'être une connaissance des hommes et une expérience de la vie pour devenir une science, est-elle restée la plus vaine et la plus creuse des disciplines philosophiques et, dans son vide absolu, l'apanage exclusif des demi-savants et des systématiseurs stériles ? La raison est facile à trouver. La psychologie empirique a le malheur de n'avoir pas même un objet, au sens scientifique d'une technique quelconque. En posant et résolvant des problèmes, elle lutte contre le vent et les fantômes [...] Les méthodes critiques — « discursives » — ne concernent que le monde de la nature. On pourrait plutôt disséquer un thème beethovénien avec un bistouri et un acide que l'esprit par les moyens de la pensée abstraite. Connaissance de la nature et connaissance de l'homme n'ont rien de commun quant au but, au procédé et à la méthode. » (Spengler 1948)

41 Il est d'ailleurs significatif que des philosophes de la réputation de John Searle s'intéressent à l'IA. En France, les cas sont plus rares ; certes, Jean Largeault et les membres de son école s'intéressent à la philosophie des sciences, mais l'intérêt pour l'informatique reste, hélas, très limité.

faut-il pouvoir s'appuyer sur des expériences et des phénomènes concrets, développer des théories vérifiables et falsifiables, obtenir des résultats tangibles. Or, le discours du modèle cognitif symbolique s'appuie, certes, sur des résultats tangibles, les résultats de vingt ans de recherche en IA (langages de programmation logique, systèmes experts, algorithmes de recherche dans les graphes...) ; mais ces résultats n'ont pas grand chose à voir avec ce que les partisans des modèles cognitifs prétendaient construire. On est arrivé à une situation, finalement assez ordinaire dans l'histoire des sciences, où l'on a trouvé autre chose que ce que l'on cherchait. Il reste maintenant à exploiter correctement cet « autre chose », à s'intéresser aux problèmes pragmatiques qui s'offrent à nous, et à accepter de faire une sévère auto-critique[42] afin de recentrer la recherche à la lumière des résultats obtenus, pour la rendre plus efficace. Cela entraînera peut-être des révisions déchirantes, mais nécessaires.

23.5 Addendum à l'édition 2002

En écrivant la première édition de ce livre, il y a dix ans, un de nos buts était de dénoncer une dérive qui nous semblait dangereuse. Aujourd'hui, le débat qui faisait rage s'est quelque peu éteint. Si l'approche cognitive de l'IA a encore quelques partisans, ils se font de plus en plus rares. De façon encore plus symptomatique, les mots « Intelligence Artificielle » tendent à disparaître lentement du vocabulaire de la recherche scientifique. Victime d'un redoutable mouvement de balancier, les équipes qui se réclamaient de cette discipline il y a quelques années préfèrent souvent aujourd'hui habiller leurs activités d'autres termes.

De la même façon que nous redoutions les excès de l'approche cognitive, nous nous méfions également des erreurs d'interprétation et des exagérations qui se font maintenant parfois jour au sein des écoles « connexionistes » ou « évolutionnistes ». La possibilité de développer des machines intelligentes ou de résoudre tous les problèmes à l'aide de réseaux de neurones ou d'algorithmes génétiques n'est pas, aujourd'hui, plus vraisemblable que ne l'étaient hier les mêmes discours dans le domaine de l'IA cognitive. Un « réseau de neurones à apprentissage supervisé » n'est pas autre chose qu'une composition de fonctions linéaires et d'une fonction non linéaire monotone dont on cherche le minimum par une méthode de gradient ; cette seconde vision est sans doute moins poétique, mais elle aide à se rappeler que nos modèles mathématiques sont très loin de la réalité physiolo-

42 Il faut remarquer que cette auto-critique est entreprise au sein de l'IA depuis déjà quelques années. Un des exemples les plus classiques est certainement l'article de Ritchie et Hanna (Ritchie and Hanna 1984), s'attaquant au travail de Lenat (Lenat 1976) sur AM (rappelons brièvement que AM est un système expert qui avait pour but de découvrir des concepts et des théorèmes mathématiques). L'article de Ritchie et Hanna note dans le travail de Lenat « d'inquiétantes différences entres les résultats théoriques annoncés et le programme réellement implanté ». Ils notent également que : « les compte-rendus écrits [...] donnent une vue trompeuse de la façon dont le programme fonctionne ». Ritchie et Hanna mirent également en doute la méthodologie employée par Lenat : ils mirent en doute la possibilité de reproduire les résultats de Lenat, et émirent même l'hypothèse que des interventions ad hoc de l'auteur du programme firent bien plus pour l'obtention des résultats que le processus automatique lui-même. Ce type de critique pourrait sans doute s'appliquer à bien des programmes d'IA encore aujourd'hui, spécialement dans le champ de la recherche cognitive (une des raisons de la publication de cet article dans *Artificial Intelligence*, qui est la grande revue de l'Intelligence Artificielle, était précisément le caractère « généralisable » du travail de Ritchie et Hanna).

gique. On pourrait en dire autant des algorithmes génétiques, dont l'auteur de ces lignes est pourtant un fervent utilisateur.

Il n'y a ni méthode universelle, ni solution unique. L'informatique moderne offre un vaste ensemble d'outils puissants, parmi lesquels il faut savoir pragmatiquement choisir celui qui sera le plus adapté au problème rencontré. C'est bien plus souvent dans la modélisation du problème et dans le choix de l'outil que se trouve la partie utile et créative du travail que dans le développement de techniques miraculeuses.

Les chevaliers de la table ronde eurent leur quête du Graal, les alchimistes celle de la pierre philosophale, toutes vouées à un échec certain. Peut-être que dans quelques siècles le rêve de développer des machines intelligentes apparaîtra-t-il aussi comme une recherche sans but. Si le rêve reste une source de motivation indispensable à l'avancée scientifique, il faut sans cesse tenter de garder objectivité, rigueur scientifique et prudence. *In medio stat virtus.*

Bibliographie essentielle

[1] Jacques Arsac. *Les machines à penser.* Seuil, 1987. *Une réflexion sur l'intelligence artificielle et les ordinateurs par un des pionniers de l'informatique en France.*

[2] Harold Abelson, Gerald Jay Sussman, et Julie Sussman. *Structure et interprétation des programmes informatiques.* InterÉditions, 1989. Traduction de (Abelson *et al.* 1985). *Une introduction à l'informatique via* SCHEME. *Chaudement recommandé.*

[3] J.P. Barthélemy, G. Cohen, et A. Lobstein. *Complexité algorithmique et problèmes de communications.* Masson – collection technique et scientifique des télécommunications, Paris, France, 1992. ISBN : 2-225-82672-2. *Une introduction à la théorie de la complexité suivie de quelques applications à des problèmes de communication.*

[4] Danièle Beauquier, Jean Berstel, et Philippe Chrétienne. *Éléments d'algorithmique.* Masson (Manuels informatiques), Paris, France, 1992. ISBN : 2-225-82704-4. *Des algorithmes et des structures de données raffinées sur les arbres et les graphes...*

[5] J. H. Conway, E. Berkelamp, et R.K. Guy. *Winning Ways (for your mathematical plays).* Academic Press, New York, 1982. ISBN : 0-12-091150-7. *Une « introduction » en douceur aux jeux de Conway, vus façon BCG. Une petite merveille pour vos longues soirées d'hiver, une source inépuisable d'idées de programmes.*

[6] Gregory Chaitin. *Algorithmic Information Theory.* Cambridge University Press, 1987. ISBN : 0-521-34306-2. *Un ouvrage de référence sur la complexité au sens de Chaitin-Kolmogorov.*

[7] Thomas H. Cormen, Charles E. Leiserson, et Ronald L. Rivest. *Introduction to algorithms.* MIT Press, 1990. ISBN : 0-262-03141-8. *Encore des algorithmes, des structures de données... Un livre très complet (plus de 1000 pages) dans lequel vous ne pouvez pas ne pas trouver votre bonheur.*

[8] Jean-Paul Delahaye. *Outils logiques pour l'intelligence artificielle.* Eyrolles, 1986. *Logiques classiques et décidabilité : une introduction claire et précise.*

[9] Didier Dubois et Henri Prade. *Théorie des possibilités.* Masson, 1988. *Tout sur le flou et la théorie des possibilités.*

[10] Hubert Dreyfus. *Intelligence Artificielle, mythes et limites.* Flammarion, 1984. *Une critique bien construite de l'IA. La traduction française correspond à la deuxième édition américaine (Dreyfus 1979).*

[11] Henri Farreny et Malik Ghallab. *Éléments d'intelligence artificielle.* Hermès, 1987. *Une présentation générale de l'IA accompagnée d'une analyse assez poussée des algorithmes d'exploration style* A^*_ε.

[12] Anthony J. Field et Peter G. Harisson. *Functionnal Programming.* Adisson & Wesley Publishing company, 1988. ISBN : 0-201-19249-7. *Une présentation détaillée de la programmation fonctionnelle typée et de ses principaux représentants.*

[13] Michel R. Garey et David S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-completeness.* W.H. Freeman and Company, San Francisco, 1979. ISBN : 0-7167-1044-7, 0-7167-1045-5. *La bible, un peu ancienne, mais tellement agréable à lire, pour tous les techniciens de la complexité.*

[14] Paul Gochet et Pascal Gribomont. *Logique.* Hermes, 1990. ISBN : 2-86601-249-6. *Les logiques classiques présentées dans leur contexte philosophique et historique. Comprend la présentation de nombreux algorithmes pour la preuve ou la satisfaction. Un deuxième tome traite de logiques non standards.*

[15] David Goldberg. *Genetic Algorithms.* Addison Wesley, 1989. ISBN : 0-201-15767-5. *Une présentation didactique et complète des techniques de base et des applications des algorithmes génétiques.*

[16] G. E. Hughes et M. J. Cresswell. *An Introduction to Modal Logics.* Methuen & Co. Ltd, 2nd édition, 1972. ISBN : 0-416-29460-X. *Une des bibles des « Knights of Modal Logic ».*

[17] John Hertz, Anders Krogh, et Richard G. Palmer. *Introductions to the theory of neural computing.* Addison Wesley, 1991. ISBN : 0-201-51560-1. *Une excellente introduction aux réseaux de neurones.*

[18] J. Roger Hindley et Jonathan P. Seldin. *Introduction to Combinators and* λ*-Calculus.* Cambridge University Press, 1986. ISBN : 0-521-26896-6, ISBN : 0-521-31839-4 (pbk.). *Logique combinatoire et* λ*-calcul. Une présentation didactique des deux mondes. Voir aussi (Krivine 1990) en français et (Barendregt 1981) pour aller plus loin.*

[19] Stephen Cole Kleene. *Logique mathématique.* Jacques Gabay, 1987. Édition originale : (Kleene 1967). *Les logiques classiques, décidabilité... Une présentation un peu ancienne (dans les notations et l'approche), mais didactique et complète.*

[20] Donald E. Knuth. *The art of computer programming,* trois volumes actuellement publiés. Addison Wesley. ISBN : 0-201-03809-9, 0-201-03822-6, 0-201-03803-X. *Trois gros volumes présentant les techniques algorithmiques de base.*

[21] Yves Kodratoff. *Leçons d'Apprentissage Symbolique Automatique.* Cepadues, Toulouse, France, 1986. *Une présentation générale de l'apprentissage symbolique.*

[22] John R. Koza. *Genetic programming.* MIT Press, 1992. ISBN : 0-262-11170-5. *Une présentation tout-à-fait exhaustive (plus de 800 pages) des techniques de programmation génétique, accompagnée de nombreux exemples d'application.*

[23] R. Lalement. *Logique, Réduction, Résolution.* Masson, Études et Recherches en informatique, 1990. *Une présentation synthétique autour de la logique et de la réécriture.*

[24] J. W. Lloyd. *Foundations of logic programming.* Springer Verlag, 1987. 2nd, extended ed. ISBN : 0-387-18199-7. *Les principes de la programmation logique. Précis et concis.*

[25] T. Anthony Marsland et Jonathan Schaeffer. *Computers, Chess and Cognition.* Springer-Verlag, 1990. ISBN : 0-387-97415-6. *Une collection d'articles de très bonne qualité sur les échecs et les algorithmes de jeux ainsi qu'une réflexion sur la place de la théorie des jeux dans l'intelligence artificielle.*

[26] N. J. Nilsson. *Principes d'intelligence artificielle.* Cepadues, Toulouse, France, 1988. Édition originale : (Nilsson 1986). *Une bonne présentation de l'IA, assez complète.*

[27] Judea Pearl. *Heuristique - Stratégies de recherche intelligentes pour la résolution de problèmes par ordinateur.* Cepadues Éditions, 1990. *La référence en matière de recherche dans les arbres et les graphes (y compris les jeux). Traduction parfois douteuse de (Pearl 1985).*

[28] Elaine Rich. *Intelligence Artificielle.* Masson, 1987. *Très bon livre de présentation de l'IA. L'édition originale américaine date de 1980.*

[29] Arto Salomaa. *Introduction à l'informatique théorique.* Armand Colin, 1989. Version française de (Salomaa 1985). *Les thèmes essentiels de l'informatique théorique : théorie des langages, automates, calculabilité, complexité, cryptographie...*

[30] André Thaysse et al. *Approche logique de l'intelligence artificielle, Tome I : de la logique classique à la programmation logique.* Dunod, 1988. *Les logiques classiques. Les tomes suivants traitent des logiques non classiques.*

[31] Edward Tsang. *Foundations of Constraint Satisfaction.* Academic Press Ltd., London, 1993. ISBN : 0-12-701610-4. *La première référence entièrement consacrée aux problèmes de satisfaction de contraintes.*

[32] Pierre Wolper. *Introduction à la calculabilité.* InterÉditions, 1991. *Présentation didactique de la théorie de la complexité et de la calculabilité.*

[33] Nguyen Huy Xuong. *Mathématiques discrètes et informatique.* Masson (Logique Mathématique Informatique), Paris, France, 1992. ISBN : 2-225-82621-8. *Rappels de mathématiques : inductions, ordres, treillis, plus petit point fixe... Informatique : théorie de la complexité, graphes...*

Bibliographie

[1] Emile Aarts and Jan Karel Lenstra. *Local Search in Combinatorial Optimization.* Interscience Series in Discrete Mathematics and Optimization. John Wiley and sons, 1997.

[2] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and interpretation of computer programs.* MIT Press, 1985. ISBN : 0-262-01077-1.

[3] Adelson-Velsky, Arlazarov, and Donskoy. *Algorithms for games.* Springer-Verlag, 1952. ISBN : 0-387-96629-3.

[4] M. S. Affane and H. Bennaceur. A weighted arc consistency technique for Max-CSP. In *Proc. of the 13th ECAI*, pages 209–213, Brighton, United Kingdom, 1998.

[5] Hassan Ait-Kaci. *The Warren abstract machine : a tutorial reconstruction.* MIT press, 1991. ISBN : 0-262-01123-9, 0-262-51058-8.

[6] S. G. AKL, D. T. Barnard, and R. J. Doran. Design, analysis and implementation of a parallelel tree search. *IEEE-PAMI*, 4(2), 1982.

[7] J. S. Albus. A new approach to manipulator control : The cerebellar model articulation controller (CMAC). *Journal of Dynamic Systems, Measurement and Control*, 97 :270–277, 1975.

[8] Daniel Alkon. Mémorisation et neurones. *Pour la science*, 143, Septembre 1989.

[9] J. F. Allen. An interval-based representation for temporal knowledge. In *7th International Conference on Artificial Intelligence, Vancouver, Canada*, 1981.

[10] Jean-Marc Alliot and Andreas Herzig. Implementing PROLOG extensions : a parallel inference machine. In *Proceedings of Fifth Generation Computer Systems conference (FGCS-92)*, volume 2, pages 833–842. Institute for New Generation Computing Technology, OHMSA, Ltd, June 1992.

[11] Jean-Marc Alliot and Thomas Schiex. *Intelligence Artificielle et Informatique Théorique.* Cepadues, 1993. ISBN : 2-85428-324-4.

[12] Jean-Marc Alliot. *TARSKI, Une machine parallèle pour l'implantation d'extensions de PROLOG.* Thèse de doctorat, Université Paul Sabatier, 1992.

[13] T. S. Anantharaman, M.S. Campbell, and F. H. Hsu. Singular extensions : Adding selectivity to brute-force searching. *Artificial Intelligence*, 43(1) :99–110, 1990.

[14] David Applegate, Guy Jacobson, and Daniel Sleator. Computer analysis of Sprouts. Technical Report (CMU–CS–91–144), Carnegie Mellon University, Pittsburgh, PA 15213, May 1991.

[15] William Armstrong and Jan Gecsi. Adaptation algorithms for binary tree networks. *IEEE transactions on systems, man and cybernetics*, 9(5), 1979.

[16] William Armstrong, Jiandong Liang, Dekang Lin, and Scott Reynolds. Experiments using parsimonious adaptative logic. Technical Report TR 90-30, University of Alberta, Edmonton, Alberta, Canada, 1990.

[17] William Armstrong, Jiandong Liang, Dekang Lin, and Scott Reynolds. Experience using adaptative logic networks. In *Proceedings of the IASTED International Symposium on Computers, electronics, communication and control*, 1991.

[18] William Armstrong. Adaptive boolean logic element. US patent 3 934 231, January 1976.

[19] Jacques Arsac. *Les machines à penser*. Seuil, 1987.

[20] H. Asada and S. Liu. Transfer of Human Skills to Neural Net Robot Controllers. In *Proceeding of IEEE conference on Robotics and Automation*, Sacramento, 1991. IEEE.

[21] Philippe Jégou Assef Chmeiss. Path-consistency : When space misses time. In AAAI Press, editor, *Proc. of AAAI-96*, pages 196–201, 1996.

[22] Eric Audureau, Patrice Enjalbert, and Luis Fariñas Del Cerro. *Logique temporelle. Sémantique et validation des programmes parallèles*. Masson, 1990.

[23] M. Bacharach and S. Hurley, editors. *Foundations of Decision Theory*, chapter Dynamic Consistency and Non-Expected Utility, by Machina, M. J., pages 39–91. Basil-Blackwell, Oxford, 1991.

[24] T. Bäck and H. P. Schwefel. An overview of evolutionary algorithms for parameter optimization. *Evolutionary Computation*, 1(1) :1–24, 1993.

[25] T. Back, G. Rudolph, and H. P. Schwefel. Evolutionary programming and evolution strategies : similarities and differences. In D. B. Fogel and W. Atmar, editors, *Proceedings of the* 2nd *Annual Conference on Evolutionary Programming*, pages 11–22. Evolutionary Programming Society, 1993.

[26] J. Backus. Can programming be liberated from the von neumann style ? a functionnal style and its algebra of programs. *ACM*, 21(8) :613–641, 1978.

[27] Doug Bagley. The great computer language shootout. Web Page, 2001. www.bagley.org/~doug/shootout/.

[28] L. Baird and A. Moore. Gradient descent for general reinforcement learning. *NIPS-12*, 1999.

[29] H.P. Barendregt. *The λ-Calculus, Its Syntax and Semantics*. North Holland Publishing Company, Amsterdam, Netherlands, 1981. La bible des « Knights of Lambda-Calculus ». ISBN : 0-444-87508-5.

[30] J.P. Barthélemy, G. Cohen, and A. Lobstein. *Complexité algorithmique et problèmes de communications*. Masson – collection technique et scientifique des télécommunications, Paris, France, 1992. ISBN : 2-225-82672-2.

[31] A.G. Barto, S.J. Bradtke, and S.P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72 :81–138, 1995.

[32] Marianne Baudinet. Temporal logic programming is complete and expressive. In *Sixteenth ACM Symposium on Principles of Programming Language*, 1989.

[33] J. Baxter and P. L. Bartlett. Reinforcement learning in pomdp's via direct gradient ascent. In *International Conference on Machine Learning*, volume 17, 2000.

[34] J. Baxter, A. Tridgell, and L. Weaver. Knightcap : a chess program that learns by combining td(λ) with game-tree search. In *International Conference on Machine Learning*, volume 15, 1998.

[35] John D. Beasley. *The mathematics of games*. Oxford University Press, 1990. ISBN : 0-19-286107-7.

[36] Danièle Beauquier, Jean Berstel, and Philippe Chrétienne. *Éléments d'algorithmique*. Masson (Manuels informatiques), Paris, France, 1992. ISBN : 2-225-82704-4.

[37] Y. Bekkers, B. Canet, O. Ridoux, and L. Ungaro. A memory management machine for Prolog interpreters. In S-Å. Tärnlund, editor, *2nd Int. Conf. Logic Programming*, pages 343–351. Uppsala University, 1984.

[38] Richard E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[39] Belaid Benhamou and Lakhdar Sais. Formules de cardinalité et symétries en calcul propositionnel. In *Actes des premières rencontres jeunes chercheurs en IA*, Rennes, France, September 1992.

[40] H.R. Berenji. Fuzzy Q-learning : A new approach for fuzzy dynamic programming problems. In *Proceedings of the 3th IEEE Fuzzy systems conference*, Orlando, FL, June 94.

[41] Claude Berge. *Graphes et hypergraphes*. Dunod, Paris, France, 1970.

[42] Claude Berge. *Hypergraphes — Combinatoire des ensembles finis*. Gauthiers-Villars, France, 1987.

[43] P. Berlandier and B. Neveu. Maintaining Arc Consistency through Constraint Retraction. In *Proc. of the 6th IEEE International Conference on Tools with Artificial Intelligence (TAI94)*, New Orleans, LA, 1994.

[44] P. Berlandier. Intégration d'outils pour l'expression et la satisfaction de contraintes dans un générateur de systèmes experts. Technical Report 924, Rapport de recherche INRIA, Programme 1, November 1988.

[45] Pierre Berlandier. Filtrage de problèmes par consistance de chemin restreinte. *Revue d'Intelligence Artificielle*, 9(3) :225–238, 1995.

[46] Hans Berliner. L'ordinateur champion de backgammon. *Pour la science*, Août 1980.

[47] Hans Berliner. Hitech. In *Computers, Chess and Cognition*. Springer-Verlag, 1990. In (Marsland and Schaeffer 1990).

[48] Pierre Berloquin. *100 jeux logiques*. Le livre de poche, 1973.

[49] Umberto Bertelé and Francesco Brioshi. *Nonserial Dynamic Programming*. Academic Press, 1972.

[50] D. P. Bertsekas and J. N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, Belmont (MA), 1996.

[51] D. P. Bertsekas. *Dynamic Programming : Deterministic and Stochastic Models*. Prentice, 1987.

[52] Christian Bessière and Marie-Odile Cordier. Arc-consistency and arc-consistency again. In *Proc. of AAAI-93*, Washington, DC, July 1993.

[53] Christian Bessière and Jean-Charles Régin. Mac and combined heuristics : Two reasons to forsake fc (and cbj ?) on hard problems. In *Proc. of the Second International Conference on Principles and Practice of Constraint Programming*, pages 61–75, Cambridge (MA), August 1996.

[54] Christian Bessière and Jean-Charles Régin. Arc consistency for general constraints networks : preliminary results. In *Proc. of the 15th IJCAI*, pages 398–404, Nagoya, Aichi, Japan, 1997.

[55] Christian Bessière and Jean-Charles Régin. Refining the basic constraint propagation mechanism. In *Proc. of the 17th IJCAI*, pages 309–315, Seattle, WA, 2001.

[56] C. Bessière, E.C. Freuder, and J.C. Régin. Using inference to reduce arc-consistency computation. In *Proc. of the 14th IJCAI*, Montréal, Canada, August 1995.

[57] Christian Bessière. Les CSP au secours des TMS. In *Actes des Journées PRC-IA, Pôles A-E*, pages 335–364, Plestin-les-Grèves (Côtes d'Armor), September 1991.

[58] Christian Bessière. Arc-consistency for non-binary dynamic CSPs. In *Proc. of the 10th ECAI*, pages 23–27, Vienna, Austria, August 1992.

[59] Christian Bessière. *Systèmes à contraintes évolutifs en Intelligence Artificielle.* Thèse de doctorat, Université des Sciences et Techniques du Languedoc, Montpellier, France, September 1992.

[60] S. Bhattacharya and A. Bagchi. Making use of available memory when searching game trees. In *Proc. of AAAI-86*, Philadelphia, PA, August 11–15 1986. Présentation de l'IterSSS*.

[61] S. Bistarelli, U. Montanari, and F. Rossi. Constraint solving over semirings. In *Proc. of the 14th IJCAI*, Montréal, Canada, August 1995.

[62] S. Bistarelli, H. Fargier, U. Montanari, F. Rossi, T. Schiex, and G. Verfaillie. Semiring-based csps and valued csps : Basic properties and comparison. In M. Jampel, E. Freuder, , and M. Maher, editors, *Over-Constrained Systems*, number 1106 in LNCS, pages 111–150. Springer Verlag, 1996.

[63] P. Boizumault. *PROLOG, l'implantation.* MASSON, 1988.

[64] P. Bourret, J. Reggia, and M. Samuelides. *Réseaux neuronaux.* Teknea, 1991. ISBN : 2977100160.

[65] J. Boyan and M. Littman. Packet routing in dynamically changing networks : A reinforcement learning approach. In J. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems 6.* Morgan Kaufmann, 1993.

[66] Karl S. Brace, Richard R. Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In *27th ACM/IEEE Design Automation Conference*, pages 40–45, 1990.

[67] Ivan Bratko. *Programmer en PROLOG pour l'Intelligence Artificielle.* InterÉditions, 1988.

[68] Philippe Breton. Une histoire de l'informatique. *La découverte*, 1987.

[69] C. L. Bridges and D. E. Goldberg. An analysis of reproduction and crossover in a binary-coded genetic algorithm. In *Proceedings of the Second International Conference on Genetic Algorithm.* ICGA, 1987.

[70] Rodney Brooks. Intelligence without reason. Memo 1293, MIT, 1991. Also in *Proc. of IJCAI-91.*

[71] M. Bruynooghe and L.M. Pereira. Deduction revision by intelligent backtracking. In J.A. Campbell, editor, *Implementation of Prolog*, chapter « Deduction revision by intelligent backtraking », pages 194–215. Ellis Horwood, 1984.

[72] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8) :677–691, 1986.

[73] T.N. Bui and B. R. Moon. Hyperplane synthesis for genetic algorithms. In *Proceedings of the Fifth International Conference on Genetic Algorithm.* ICGA, 1993.

[74] BULL S.A. *Charme V1, Manuel de référence*, March 1990. Ref : 95 F2 22GNA Rev0.

[75] Michael Buro. Toward opening book learning. In *Proceedings of IJCAI97 workshop on computer games*, 1997.

[76] J. A. Campbell, editor. *Implementations of PROLOG.* Ellis Horwood, 1984. ISBN : 0-470-20044-8.

[77] J. J. Cannat, G. Bisson, J. Sayous, and J. Goubert. Rapport final sur la convention LRI/CENA : PERSPICACE sur l'utilisation de l'apprentissage symbolique automatique pour la résolution de conflits. Technical Report CENA R 88-21, Centre d'Études de la Navigation Aérienne, 1988.

[78] Carnegie Mellon. *First Machine Learning Workshop*, 1980.

[79] John Mc Carthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, April 1960.

[80] R. A. Caruana and J. D. Schaffer. Representation and hidden bias : Gray versus binary coding for genetic algorithms. In *Proceedings of the Fifth International Conference on Machine Learning*, 1988.

[81] Jean Cavaillès. *Philosophie mathématique.* Hermann, 1962. Recueil des écrits de Jean Cavaillès. ISBN : 2-7056-5047-4.

[82] Michel Cayrol, Olivier Palmade, and Thomas Schiex. A fixed point semantics for the ATMS. *Journal of Logic and Computation*, 3(2) :115–130, April 1993.

[83] R Cerf. *Une Théorie Asymptotique des Algorithmes Génétiques.* PhD thesis, Université Montpellier II (France), 1994.

[84] Gregory Chaitin. Série aléatoire et preuve mathématique. *Pour la science*, Juin 1979.

[85] Gregory Chaitin. *Algorithmic Information Theory.* Cambridge University Press, 1987. ISBN : 0-521-34306-2.

[86] Gregory Chaitin. Incompleteness theorem for random reals. *Advanced Applied Mathematics*, 8, 1987.

[87] Gregory Chaitin. Le hasard en théorie des nombres. *Pour la science*, 131, Septembre 1988.

[88] C. L. Chang and R. C. T. Lee. *Symbolic logic and mechanical theorem proving.* Academic press, New-York, 1987. ISBN : 0-12-170350-9.

[89] Jean-Pierre Changeux and Alain Connes. *Matière à pensée.* Odile Jacob, 1989.

[90] D. Chapman and L.P. Kaelbling. Input Generalization in Delayed Reinforcement Learning : An Algorithm and Performance Comparisons. In *IJCAI'91*, pages 726–731, 1991.

[91] P. Cheeseman, B. Kanefsky, and W. Taylor. Where the *really* hard problems are. In *Proc. of the 12th IJCAI*, pages 294–299, Sidney, Australia, 1991.

[92] K. Chen, A. Kierulf, M. Muller, and J. Nievergelt. The design and evolution of Go Explorer. In *Computers, Chess and Cognition.* Springer-Verlag, 1990. In (Marsland and Schaeffer 1990).

[93] P. Chen. Heuristic sampling : a method for predicting the performance of tree searching programs. *SIAM Journal on Computing*, 21(2) :295–315, 1992.

[94] Paul Churchland and Patricia Churchland. Les machines peuvent-elles penser ? *Pour la science*, 149, Mars 1990.

[95] P. Cichosz. Truncating temporal differences : On the efficient implementation of td(λ) for reinforcement learning. *Journal of Artificial Intelligence Research (JAIR)*, 2 :287–318, 1995.

[96] K. M. Colby, S. Weber, and F. D. Hilf. Artificial paranoia. *Artificial Intelligence*, 2 :1–25, 1971.

[97] K. M. Colby, F. D. Hilf, S. Weber, and H. C. Kraemer. Turing-like indistinguishability tests for the validation of a computer simulation of paranoid processes. *Artificial Intelligence*, 3 :199–221, 1972.

[98] A. Colmerauer. Notes sur PrologIII. In *Séminaire sur la programmation logique*, Tregastel, May 1986.

[99] J. H. Conway, E. Berkelamp, and R.K. Guy. *Winning Ways (for your mathematical plays).* Academic Press, New York, 1982. ISBN : 0-12-091150-7.

[100] John Horton Conway. *On Numbers and Games.* Academic Press, 1976. ISBN : 0-12-186350-6.

[101] M. C. Cooper. An optimal k-consistency algorithm. *Artificial Intelligence*, 41 :89–95, 1989.

[102] Martin C. Cooper. Characterising tractable constraints. *Artificial Intelligence*, 65 :347–361, 1994.

[103] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to algorithms.* MIT Press, 1990. ISBN : 0-262-03141-8.

[104] Guy Cousineau and Gérard Huet. The CAML Primer, projet Formel. Technical report, INRIA – ENS, 1989.

[105] G. Cousineau, P. L. Curien, and M. Mauny. The categorial abstract machine. In J. Jouannaud, editor, *Functionnal Programming Languages and Computer Architecture*, volume 201, pages 50–64. LNCS, 1985.

[106] R. Crites and A. Barto. Improving elevator performance using reinforcement learning. In *Neural Information Processing Systems (NIPS)*, 1996.

[107] Luis Damas and Robin Milner. Principal types schemes for functionnal programs. *9th ACM Symp. on the Principles of Prog. Lang.*, pages 207–211, 1982.

[108] D. Dasgupta and D. R. McGregor. A structured genetic algorithm. Technical Report IKBS-8-92, Dep. of Computer Science. University of Strathclyde, Glasgow. UK, 1992.

[109] E. Davalo and P. Naim. *Des réseaux de neurones.* Eyrolles, 1986.

[110] Philippe David. Quelques propriétés des consistances d'arc et de chemin pour deux classes de problèmes de satisfaction de contraintes. In *Actes des rencontres des jeunes chercheurs en IA*, Rennes, France, September 1992.

[111] Philippe David. When functional and bijective constraints make a CSP polynomial. In *Proc. of the 13th IJCAI*, Chambery, France, August 1993.

[112] T. E. Davis and J. C. Principe. A simulated annealing like convergence theory for simple genetic algorithm. In R.K. Belew and L.B. Booker, editors, *Proceedings of the 4th International Conference on Genetic Algorithms*, pages 174–181. Morgan Kaufmann, 1991.

[113] T. E. Davis and J. C. Principe. A markov chain framework for simple genetic algorithm. In *Evolutionary Computation*, volume 1 (3), pages 269–292, 1993.

[114] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3) :210–215, 1960.

[115] M. Davis. Hilbert's tenth problem is unsolvable. *American mathematical monthly*, 80, 1973.

[116] M. Davis. *Computability and unsolvability.* Dover, New-York, 1982. ISBN : 0-486-61471-9 Première édition : 1958.

[117] Randall Davis. Expert Systems : Where Are We ? And Where Do We Go From Here ? *The AI Magazine*, Printemps 1982.

[118] Randall Davis. Diagnostic reasoning based on structure and behavior. *Artificial Intelligence*, 24, 1984.

[119] E. Davis. Constraint propagation with interval labels. *Artificial Intelligence*, 32(3) :281–331, July 1987.

[120] Randall Davis. Issues in model based troubleshooting. Technical Report 893, MIT, 1987.

[121] Richard Dawkins. *The blind watchmaker.* Norton, New York, 1986. ISBN : 0-393-02216-1.

[122] Richard Dawkins. *L'horloger aveugle.* Robert Laffont, 1989. Edition originale : (Dawkins 1986).

[123] Johan de Kleer. An assumption-based truth maintenance system. *Artificial Intelligence*, 28 :127–162, 1986.

[124] Johan de Kleer. Extending the ATMS. *Artificial Intelligence*, 28 :163–196, 1986.

[125] Johan de Kleer. Problem solving with the ATMS. *Artificial Intelligence*, 28 :197–224, 1986.

[126] J. de Kleer. A comparison of ATMS and CSP techniques. In *Proc. of the 11th IJCAI*, pages 290–296, Detroit, MI, August 1989.

[127] Romuald Debruyne and Christian Bessière. From restricted path consistency to max-restricted path consistency. In *Proc. of CP'97*, number 1330 in LNCS, pages 312–326, Linz, Austria, November 1997. Springer-Verlag.

[128] Romuald Debruyne and Christian Bessière. Some practicable filtering techniques for the constraint satisfaction problem. In *Proc. of the 15th IJCAI*, pages 412–415, Nagoya, Aichi, Japan, 1997.

[129] Romuald Debruyne. Les algorithmes d'arc-consistance dynamiques. *Revue d'intelligence artificielle*, 9(3) :239–267, 1995.

[130] Rina Dechter and Itay Meiri. Experimental evaluation of preprocessing techniques in constraint satisfaction problems. In *Proc. of the 11th IJCAI*, pages 271–277, Detroit, MI, August 1989.

[131] Rina Dechter and Judea Pearl. The cycle-cutset method for improving search performance in AI. In *Proc. of the 3rd IEEE Conference on AI Applications*, pages 224–230, Orlando FL, 1987.

[132] Rina Dechter and Judea Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34 :1–38, 1988. See also (Kanal and V.Kumar 1988).

[133] Rina Dechter and Judea Pearl. The optimality of A*. In L. Kanal and V. Kumar, editors, *Search in Artificial Intelligence*, chapter 5, pages 166–199. Springer-Verlag, 1988.

[134] Rina Dechter and Judea Pearl. Tree clustering schemes for constraint processing. In *Proc. of AAAI-88*, pages 150–154, St. Paul, MN, 1988.

[135] Rina Dechter and Judea Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38 :353–366, 1989.

[136] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49 :61–95, 1991.

[137] Rina Dechter. Enhancement schemes for constraint processing : Backjumping, learning and cutset decomposition. *Artificial Intelligence*, 41(3) :273–312, 1990.

[138] Rina Dechter. From local to global consistency. *Artificial Intelligence*, 55 :87–107, 1992.

[139] Daniel Delahaye, Jean-Marc Alliot, Marc Schoenauer, and Jean-Loup Farges. Genetic algorithms for partitionning airspace. In *Proc. of CAIA'94*, San Antonia, Texas, March 1994.

[140] Jean-Paul Delahaye. *Outils logiques pour l'intelligence artificielle*. Eyrolles, 1986.

[141] Jean-Paul Delahaye. Le réalisme en mathématique et en physique. *Pour la science*, 159, Janvier 1991.

[142] F. D'Epenoux. A probabilistic production and inventory problem. *Management Science*, 10 :98–108, 1963.

[143] Bernard d'Espagnat. The quantum theory and reality. *Scientific American*.

[144] Bernard d'Espagnat. *Penser la science*. Dunod, 1989.

[145] Y. Deville, O. Barette, and P. Van Hentenryck. Constraint satisfaction over connected row convex constraints. In *Proc. of the 15th IJCAI*, pages 405–410, Nagoya, Aichi, Japan, 1997.

[146] W. Diffie and M. Hellman. New directions in cryptography. *IEEE transactions on information theory*, 22 :644–654, 1976.

[147] W. Dowling and J. H. Gallier. Linear time algorithm for testing the satisfiability of propositional Horn formulae. *Journal of logic programming*, 1, 1984.

[148] Jon Doyle. A truth maintenance system. *Artificial Intelligence*, 12 :231–272, 1979.

[149] Brigitta Dresp. Le traitement des illusions visuelles. *Pour la science*, Août 1991.

[150] Hubert Dreyfus and Stuart Dreyfus. *Mind over machine : the power of human intuition and expertise in the era of the computer.* New York Free Press, 1986. ISBN : 0-02-908060-6.

[151] Hubert Dreyfus. *What computers can't do : the limits of artificial intelligence.* Harper and Row, 1979. ISBN : 0-06-090624-3.

[152] Hubert Dreyfus. *Intelligence Artificielle, mythes et limites.* Flammarion, 1984. La traduction française correspond à la deuxième édition américaine (Dreyfus 1979).

[153] Dominique Dubarle. Critique du réductionnisme. In Jean Piaget, editor, *Logique et connaissance scientifique.* Encyclopédie de la Pléiade, Gallimard, 1967.

[154] Didier Dubois and Henry Prade. *Théorie des possibilités.* Masson, 1988.

[155] Catherine Dubois, François Rouaix, and Pierre Weis. Extensional polymorphism. In *Proceedings of the 22th ACM Conference on Principles of Programming Languages*, January 1995.

[156] Jean-Jacques Duby. Point de vue. *Pour la science*, Juillet 1990.

[157] Jean-Paul Dumont. *Les écoles pré-socratiques.* Folio, 1991. Recueil commenté (presque) extensif des textes des philosophes pré-socratiques parvenus jusqu'à nous. Une édition plus largement annotée existe dans l'Encyclopédie de la Pléiade.

[158] Florence Dupin de Saint Cyr, Jérôme Lang, and Thomas Schiex. Gestion de l'inconsistance dans les bases de connaissances : une approche syntaxique basée sur la logique des pénalités. In *Actes de RFIA'94*, pages 507–518, France, 1994.

[159] Nicolas Durand and Jean-Marc Alliot. A combined Nelder-Mead simplex and genetic algorithm. In *Proceedings of GECCO'99*, 1999.

[160] Nicolas Durand. *Optimisation de trajectoires pour la résolution de conflits aériens en route.* PhD thesis, Institut National Polytechnique de Toulouse, Mai 1996.

[161] A. Dutech. *Apprentissage d'environnement : approches cognitives et comportementales.* Thèse de Doctorat de l'École Nationale Supérieure de l'Aéronautique et de l'Espace, Toulouse, 1999.

[162] Peter Elmas. La perception de la parole par les nourrissons. *Pour la science*, 89, Mars 1985.

[163] Herbert B. Enderton. *A mathematical introduction to logic.* Academic Press, Inc., 1972. ISBN : 0-12-238450-4.

[164] Pascal Engel. *La norme du vrai. Philosophie de la logique.* NRF Essais. Gallimard, 1989.

[165] P. Enjalbert and L. Fariñas del Cerro. Modal resolution in clausal form. *Theoretical Computer Science*, 65, 1989.

[166] G. Escalada and M. Ghallab. A pratically efficient and almost linear unification algorithm. Rapport de recherche 8727, LAAS, Toulouse, France, Janvier 1987.

[167] Robert Escarpit. *Théorie de l'information et pratique politique.* Seuil, 1981.

[168] Xavier Leroy et Pierre Weis. *Manuel de Référence du langage Caml.* InterEditions, 1993.

[169] H. Fargier and J. Lang. Uncertainty in constraint satisfaction problems : a probabilistic approach. In *Proc. of ECSQARU '93, LNCS 747*, pages 97–104, Grenade, Spain, November 1993.

[170] Hélène Fargier, Roger Martin-Clouaire, and Thomas Schiex. Satisfaction de contraintes souples. In *Journées nationales « Les applications des ensembles flous »*, pages 183–191, Nîmes, France, October 1992.

[171] H. Fargier, J. Lang, and T. Schiex. Selecting preferred solutions in Fuzzy Constraint Satisfaction Problems. In *Proc. of the 1st European Congress on Fuzzy and Intelligent Technologies*, 1993.

[172] Hélène Fargier. Problèmes de satisfaction de contraintes floues. Technical Report 92-29-R, IRIT-UPS, Toulouse, France, September 1992.

[173] L. Fariñas del Cerro. A simple deduction method for modal logic. *Information Processing Letters*, 14(2), 1982.

[174] Henri Farreny and Malik Ghallab. *Éléments d'intelligence artificielle.* Hermès, 1987.

[175] Edward A. Feigenbaum and Pamela McCorduck. *The Fifth Generation : Artificial Intelligence and Japan's computer challenge to the world.* Addison-Wesley, 1983. 0-201-11519-0.

[176] Edward Feigenbaum and Pamela McCorduck. *La cinquième génération.* InterEditions, 1984. Traduction de (Feigenbaum and McCorduck 1983).

[177] Anthony J. Field and Peter G. Harisson. *Functionnal Programming.* Adisson & Wesley Publishing company, 1988. ISBN : 0-201-19249-7.

[178] L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence Through Simulated Evolution.* Wiley and sons. NY, 1966.

[179] D. B. Fogel, L. J. Fogel, W. Atmar, and G. B. Fogel. Hierarchics methods of evolutionary programming. In D.B. Fogel and W. Atmar, editors, *Proceedings of the 1rst annual conference on Evolutionary Programming.* Evolutionary Programming Society, 1992.

[180] L.J. Fogel. Autonomous automata. *Industrial research*, 4 :14–19, 1962.

[181] Mark Fox. AI and Expert Systems : Myths, Legends and Facts. *IEEE Expert*, Fevrier 1990.

[182] Roland Fraïssé. *Cours de logique mathématique.* Gauthiers Villars, deuxième edition, 1971.

[183] Eugene C. Freuder and Michael J. Quinn. Taking advantage of stable sets of variables in constraint satisfaction problems. In *Proc. of the 9th IJCAI*, pages 1076–1078, Los Angeles, CA, 1985.

[184] E.C. Freuder and R.J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58 :21–70, December 1992.

[185] E. C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21 :958–966, November 1978.

[186] Eugene C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1) :24–32, 1982.

[187] Eugene C. Freuder. A sufficient condition for backtrack-bounded search. *Journal of the ACM*, 32(14) :755–761, 1985.

[188] Eugene C. Freuder. Backtrack-free and backtrack-bounded search. In L. Kanal and V. Kumar, editors, *Search in Artificial Intelligence*, chapter 10, pages 343–369. Springer-Verlag, 1988.

[189] Eugene C. Freuder. Partial constraint satisfaction. In *Proc. of the 11th IJCAI*, pages 278–283, Detroit, MI, 1989.

[190] Eugene C. Freuder. Complexity of the k-tree structured constraint satisfaction problems. In *Proc. of AAAI-90*, pages 4–9, Boston, MA, 1990.

[191] Christine Froidevaux. Taxonomic default theory. In *Artificial Intelligence*, pages 123–129. ECAI-86, 1986.

[192] D. Frost and R. Dechter. Look-ahead value ordering for constraint satisfaction problems. In *Proc. of the 14th IJCAI*, pages 572–578, Montréal, Canada, 1995.

[193] John Gabriel, Tim Lindholm, E. L. Lusk, and R. A. Overbeek. A tutorial on the Warren abstract machine for computational logic. Technical report, Argonne National Laboratory, 1985.

[194] F. Garcia and F. Serre. Efficient asymptotic approximation in temporal difference learning. In *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI 2000)*, Berlin, 2000.

[195] F. Garcia and F. Serre. From Q(λ) to Average Q-learning : Efficient implementation of an asymptotic approximation. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI 2001)*, Seattle, USA, 2001.

[196] F. Garcia, L. Péret, L. Tomasini, and G. Verfaillie. A markov decision problem approach for the deployment and the maintenance of a satellite constallation. In *Proceedings of the The 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space (iSAIRAS'01)*, Montreal, Canada, 2001.

[197] Martin Gardner. *Knotted Doughnuts and other mathematical entertainments.* W.H. Freeman and Co., 1986. ISBN : 0-7167-1799-9 (pbk.), ISBN : 0-7167-1794-8 (hard).

[198] Martin Gardner. *Time travel and other mathematical bewilderments.* W.H. Freeman and Co., 1987-1988. ISBN : 0-7167-1924-X, ISBN : 0-7167-1925-8 (pbk.).

[199] Martin Gardner. *Penrose Tiles to Trapdoor Cyphers.* W.H. Freeman and Co., 1989. ISBN : 0-7167-1987-8 (pbk.), ISBN : 0-7167-1986-X (hard).

[200] Michel R. Garey and David S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-completeness.* W.H. Freeman and Company, San Francisco, 1979. ISBN : 0-7167-1044-7, 0-7167-1045-5.

[201] J. Gaschnig. A general backtrack algorithm that eliminates most redundant tests. In *Proc. of the 5th IJCAI*, page 457, Cambridge, MA, 1977.

[202] J. Gaschnig. *Performance measurement and analysis of certain Search Algorithms.* PhD thesis, Carnegie-Mellon University, 1979.

[203] F. Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques and maximum independant set of a chordal graph. *SIAM J. Comput.*, 1(2) :180–187, 1972.

[204] Pieter Andreas Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *Proc. of the 10th European Conference on Artificial Intelligence*, pages 31–35, Vienna, Austria, 1992.

[205] F. Giannesini, H. Kanaoui, R. Pasero, and M. van Caneghem. *PROLOG.* InterÉditions, 1985.

[206] Etienne Gilson. *La philosophie au Moyen Age.* Payot, 1947.

[207] Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1 :25–46, 1993.

[208] P.-Y. Glorennec. Fuzzy Q-learning and dynamical fuzzy Q-learning. In *Proceedings of the 3th IEEE Fuzzy systems conference*, Orlando, 94.

[209] Paul Gochet and Pascal Gribomont. *Logique.* Hermes, 1990. ISBN : 2-86601-249-6.

[210] G. Goetsch and M.S. Campbell. Experiments with the Null-Move Heuristic. In *Computers, Chess and Cognition.* Springer-Verlag, 1990. In (Marsland and Schaeffer 1990).

[211] D. E. Goldberg. Genetic algorithms and walsh functions. part 1 and 2. *Complex Systems*, 3 :129–171, 1989.

[212] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Reading MA Addison Wesley, 1989.

[213] David Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison Wesley, 1989. ISBN : 0-201-15767-5.

[214] M. Gordon, Robin Milner, L. Morris, M. Newey, and C. Wadsworth. A metalanguage for interactive proof in lcf. In *Proceedings of POPL'78*, pages 119–130, 1978.

[215] Julien Gracq. *La littérature à l'estomac.* José Corti, 1950.

[216] J.J. Grefenstette, C.L. Ramsey, and A.C. Schultz. Learning Sequential Decision Rules using Simulation Models and Competition. *Machine Learning*, 5(4) :355–381, 1990.

[217] Jean-Blaise Grize. Logique des classes et des propositions. In Jean Piaget, editor, *Logique et connaissance scientifique*. Encyclopédie de la Pléiade, Gallimard, 1967.

[218] Adrian De Groot. Thought and choice in chess. In Nico Frijda and Adrian De Groot, editors, *Otto Selz, his contribution to psychology*. Mouton, The Hague, 1965. ISBN : 902793438X.

[219] Noelle Bleuzen Guernalec and Alain Colmerauer. Narrowing a $2n$-block of sorting in $O(n \log n)$. In *Principles and Practice of Constraint Programming*. Springer-Verlag, 1997.

[220] M. Gyssens, P.G. Jeavons, and D. A. Cohen. Decomposing constraint satisfaction using database techniques. *Artificial Intelligence*, 66 :57–89, 1994.

[221] C.C. Han and C.H. Lee. Comments on mohr and henderson's path consistency algorithm. *Artificial Intelligence*, 36 :125–130, 1988.

[222] R. M. Haralick and G. L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14 :263–313, 1980.

[223] R. Haralick and L. Shapiro. The consistent labeling problem : Part I. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1(2) :173–184, 1979.

[224] R. Haralick and L. Shapiro. The consistent labeling problem : Part II. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2(3) :173–184, 1980.

[225] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on SSC*, 4, 1968.

[226] W. D. Harvey and M. L. Ginsberg. Limited discrepency search. In *Proc. of the* 14th *IJCAI*, Montréal, Canada, 1995.

[227] John Haugeland. *Artificial intelligence, the very idea*. MIT Press, 1985. ISBN : 0-262-08153-9.

[228] John Haugeland. *L'esprit dans la machine*. Editions Odile Jacob, 1989. Traduction de (Haugeland 1985).

[229] Bogumił Hausman. Pruning and scheduling speculative work in or-parallel Prolog. In *PARLE 89, Conference on Parallel Architectures and Languages Europe*. Springer-Verlag, 1989.

[230] Robert Hecht-Nielsen. *Neurocomputing*. Addison-Wesley, 1990. ISBN : 0-201-09355-3.

[231] G. G. Hendrix. Expanding the utility of semantic network through partitionning. In *Artificial Intelligence*. IJCAI-4, 1977.

[232] John Hertz, Anders Krogh, and Richard G. Palmer. *Introductions to the theory of neural computing*. Addison Wesley, 1991. ISBN : 0-201-51560-1.

[233] W. D. Hillis. Coevolving parasites improve simulated evolution as an optimisation procedure. In C.G. Langton, C. Taylor, J.D. Farmer, and S. Rasmussen, editors, *Artificial Life II*. Addison-Wesley, 1992.

[234] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and* λ*-Calculus*. Cambridge University Press, 1986. ISBN : 0-521-26896-6, ISBN : 0-521-31839-4 (pbk.).

[235] Geoffrey Hinton, David Plaut, and Tim Shallice. La simulation des lésions cérébrales. *Pour la science*, 194, December 1993.

[236] Keiji Hirata, Reki Yamamoto, Akira Imai, Hideo Kawai, Kiyoshi Irano, Tsuneyoshi Takagi, Kazuo Taki, and Akihiko Nakase Kazuaki Rokusawa. Parallel and distributed implementation of concurrent logic programming language KL1. In *Proceedings of FGCS'92*, 1992.

[237] Makoto Hirosawa et al. Folding simulation using temperature parallel simulated annealing. In *Proceedings of FGCS'92*, 1992.

[238] Douglas R. Hofstadter. *Metamagical Themas : questing for the essence of mind and pattern.* Penguin Books, 1990. ISBN : 0-465-04540-5.

[239] John Holland. Outline for a logical theory of adaptative systems. *Journal for the Association of Computing Machinery*, 3, 1962.

[240] Eric Hollnagel. The design of reliable interactive system. In *Cognitive Interactions, man and advanced information technologies in the control of large operational systems*, Toulouse, October 1991.

[241] Gerald Holton. *L'Invention Scientifique*, chapter « Mach, Einstein et la recherche du réel ». PUF, 1982. Une première version a été publiée en 1967.

[242] Gerald Holton. *L'Invention Scientifique*, chapter « Einstein, Michelson et l'expérience cruciale ». PUF, 1982.

[243] J. N. Hooker. A quantitative approach to logical inference. *Decision Support Systems*, 1(4) :45–69, 1988.

[244] J. Horn and N. Nafpliotis. Multiobjective optimization using the nitched pareto genetic algorithm. Illigal Report 93005, University of Illinois at Urbana, 1993.

[245] F.h Hsu, T.S. Anantharam, M.S. Campbell, and A. Nowatzyk. DEEP THOUGHT. In *Computers, Chess and Cognition.* Springer-Verlag, 1990. In (Marsland and Schaeffer 1990).

[246] J. Hu and M. Wellman. Multiagent reinforcement learning : Theoretical framework and an algorithm. In *International Conference on Machine Learning*, volume 15, 1998.

[247] Paul D. Hubbe and Eugene C. Freuder. An efficient cross-product representation of the constraint satisfaction problem search space. In *Proc. of AAAI-92*, pages 421–427, San Jose, CA, 1992.

[248] G. E. Hughes and M. J. Cresswell. *An Introduction to Modal Logics.* Methuen & Co. Ltd, 2nd edition, 1972. ISBN : 0-416-29460-X.

[249] Edmund Husserl. *Recherches logiques, Tome I : Prolégomènes à la logique pure.* PUF, 1959. Première édition : 1900, seconde édition corrigée : 1913.

[250] Edmund Husserl. *La crise des sciences occidentales et la phénoménologie transcendantale.* Gallimard, 1976. Recueil de textes d'Husserl écrits dans les années 30.

[251] N.W. Hutchings. Search for a messiah. *The Gospel Truth*, 33(2), February 1992. A Publication of Southwest Radio Church.

[252] Eero Hyvönen. Constraint reasoning based on interval arithmetic. In *Proc. of the* 11th *IJCAI*, Detroit, MI, August 1989.

[253] E. Hyvönen. Constraint reasoning based on interval arithmetic : the tolerance propagation approach. *Artificial Intelligence*, 58 :71–112, December 1992.

[254] ILOG. *Pecos, Version 1.1, Manuel de référence*, 1992. Ref :3405-MF-003.

[255] International Computers Limited. *CHIP 2.1 Built-in predicate MANUAL*, February 1990.

[256] Masato Ishikawa et al. Protein sequence analysis by parallel inference machine. In *Proceedings of FGCS'92*, 1992.

[257] Fumihide Itoh, Takashi Shikayama, Takeshi Mori, Masaki Sato, Tatsuo Kato, and Tadashi Sato. The design of the PIMOS file system. In *Proceedings of FGCS'92*, 1992.

[258] T. Jaakkola, M.I. Jordan, and S.P. Singh. On the Convergence of Stochastic Iterative Dynamic Programming Algorithms. *Neural Computation*, 6 :1185–1201, 1994.

[259] J. Jaffar and J.L. Lassez. Constraint logic programming. In *Proc. of the* 14th *ACM Symposium of the Principles of Programming Languages*, pages 111–119, Munich, January 1987.

[260] J. Jaffar, J.L. Lassez, and Maher. A logic programming language scheme. In Lindstrom DeGroot, editor, *Logic programming : Relations, Functions and Equations.* Prentice-Hall, 1985.

[261] J. Jaffar, J.L Lassez, and M.J. Maher. PrologII as an instance of the logic programming language scheme. In M. Wirsing, editor, *Formal Descriptions of Programming Concepts*. North-Holland, 1986.

[262] Joxan Jaffar, Spiro Michaylov, Peter Stuckey, and Roland Yap. The CLP($\mathcal{R}$) language and system. Technical Report RC 16292 (#72336) 11/15/90, IBM Research Division, November 1990.

[263] P. Janssen, P. Jégou, B. Nouguier, and M.C. Vilarem. A filtering process for general constraint satisfaction problems : Achieving pairewise-consistency using an associated binary representation. In *Proc. of the IEEE Workshop on Tools for Artificial Intelligence*, pages 420–427, Fairfax VA, 1989.

[264] P. Janssen. *Aide à la conception : une approche basée sur la satisfaction de contraintes*. Thèse de doctorat, Université Montpellier II, Montpellier, February 1990.

[265] P. Jégou. Cyclic-clustering : a compromise between tree-clustering and cycle-cutset method for improving search efficiency. In *Proc. of the 8th ECAI*, pages 369–371, Stockolm, Sweden, 1990.

[266] Philippe Jégou. *Contribution à l'étude des problèmes de satisfaction de contraintes : Algorithmes de propagation et de résolution - Propagation de contraintes dans les réseaux dynamiques.* Thèse de doctorat, Université des Sciences et Techniques du Languedoc, Montpellier, France, 1991.

[267] David S. Johnson. A catalog of complexity classes. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A : Algorithms and Complexity, chapter 2, pages 67–161. Elsevier, 1990.

[268] J. P. Jones and Y. Matijasevich. Register machine proof of the theorem on exponential diophantine representation of enumerable sets. *Journal of Symbolic Logic*, 49, 1984.

[269] Robin Jones. Les programmes auto-évolutifs. *Pour la science*, 177, Juillet 1992.

[270] L. Jouffe. *Apprentissage de systèmes d'inférence floue par des méthodes de renforcement : application à la régulation d'ambiance dans un bâtiment d'élevage porcin.* Thèse de Doctorat de l'Université de Rennes I, 1997.

[271] L.P. Kaelbling, M.L. Littman, and A. W. Moore. Reinforcement Learning : A Survey. *Journal of Artificial Intelligence Research (JAIR)*, 4 :237–285, 1996.

[272] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101 :99–134, 1998.

[273] L.P. Kaelbling. *Learning in Embedded Systems.* MIT Press, Cambridge, Massachusetts, 1993.

[274] H. Kaindl. Tree Searching Algorithms. In *Computers, Chess and Cognition*. Springer-Verlag, 1990. In (Marsland and Schaeffer 1990).

[275] L. Kanal and V.Kumar. *Search in Intelligence Artificial Intelligence.* Springer-Verlag, 1988. ISBN 0-387-96750-8.

[276] Emmanuel Kant. *Critique de la raison pure*, chapter « Introduction : De la différence de la connaissance pure et de la connaissance empirique », pages 35–36. Presses Universitaires de France, 1944.

[277] N. Karmarkar. A new polynomial time algorithm for linear programming. *Combinatorica*, 4 :373–395, 1984.

[278] A. Kaufman. *Introduction à la théorie des sous-ensembles flous.* Masson, 1973.

[279] Moto Kawamura, Hiroyuki Sato, Kazutomo Naganuma, and Kazumasa Yokota. Parallel management system : Kappa-P. In *Proceedings of FGCS'92*, 1992.

[280] Sœren Kierkegaard. *Les miettes philosophiques.* Editions du Seuil, 1967.

[281] H. Kimura and S. Kobayashi. Reinforcement learning for locomotion of a two-linked robot arm. In *Proceedings of the 6th European Workshop on Learning Robots*, pages 144–153, 1997.

[282] Stephen Cole Kleene. *Mathematical logic.* Wiley, New York, 1967. LC Card Number : 66026747.

[283] Stephen Cole Kleene. *Logique mathématique.* Jacques Gabay, 1987. Édition originale : (Kleene 1967).

[284] Morris Kline. *Mathematics, the loss of certainty.* Oxford University Press, 1980. ISBN : 0-19-502754-X.

[285] Morris Kline. *Mathématiques : la fin de la certitude.* Christian Bourgois éditeur, 1989. Version française de (Kline 1980).

[286] Donald E. Knuth. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29(129) :121–136, 1975.

[287] Donald E. Knuth. *The art of computer programming*, volume 2, chapter Random numbers. Addison Wesley, second edition, 1981. ISBN : 0-201-03822-6.

[288] Yves Kodratoff. *Leçons d'Apprentissage Symbolique Automatique.* Cepadues, Toulouse, France, 1986.

[289] Teuvo Kohonen. An introduction to neural computing. *Neural Networks*, 1, 1988.

[290] Teuvo Kohonen. *Self organization and associative memory.* Springer Verlag, 1988. ISBN : 0-387-51387-6.

[291] A. N. Kolmogorov. Three approaches for defining the concept of "information quantity". *Problems of Information Transmission*, 1, 1965.

[292] A. N. Kolmogorov. Logical basis for information theory and probability theory. *IEEE Transactions*, IT14, 1968.

[293] V. R. Konda and J. N. Tsitsiklis. Actor-critic algorithms. *NIPS-12*, 1999.

[294] D. Kopec. Advances in Man-Machine Play. In *Computers, Chess and Cognition.* Springer-Verlag, 1990. In (Marsland and Schaeffer 1990).

[295] R. E. Korf. Depth first iterative deepening : An optimal admissible tree search. *Artificial Intelligence*, 27 :97–109, 1985.

[296] Richard E. Korf. Real-time heuristic search : First results. In *Proc. of AAAI-87*, pages 133–138, Seattle, WA, 1987.

[297] Richard E. Korf. Real-time heuristic search : New results. In *Proc. of AAAI-88*, pages 139–144, St. Paul, MN, 1988.

[298] Richard E. Korf. Real-time heuristic search. *Artificial Intelligence*, 42 :189–211, 1990.

[299] R. Korf. Improved limited discrepancy search. In *Proc. of AAAI-96*, Portland, OR, 1996.

[300] R. Kowalsi and M. van Emden. The semantic of predicate logic as programming language. DCL Memo 73, University of Edinburgh, 1974.

[301] Alexandre Koyré. *Études d'histoire de la pensée.* Gallimard, 1966.

[302] Alexandre Koyré. *Études Galiléennes.* Hermann, 1966. Recueil de trois études publiées de 1935 à 1939.

[303] John R. Koza. *Genetic programming.* MIT Press, 1992. ISBN : 0-262-11170-5.

[304] J.L. Krivine. *λ-calcul : Types & Modèles.* Masson, Études et Recherches en informatique, 1990.

[305] Nicolai Kroguious. *La psychologie au jeu d'échecs.* Grasset - Europe Échecs, 1986. Recueil d'articles du psychologue et GMI Kroguious parus en 1967, 1968 et 1969.

[306] Thomas S. Kuhn. *The structure of scientific revolutions,.* University of Chicago Press, Chicago, 2 edition, 1970. ISBN : 0-226-45803-2.

[307] Thomas Kuhn. *La structure des révolutions scientifiques.* Flammarion, 1983. Édition originale 1962. Deuxième édition, modifiée et enrichie en 1970 (Kuhn 1970). L'édition française correspond à l'édition de 1970.

[308] Kouichi Kumon, Akira Asato, , Susumu Arai, Tsuyoshi Shinogi, Hakira Hattori, Hiriyoshi Hatazawa, and Kiyoshi Hirano. Architecture and implementation of PIM/p. In *Proceedings of FGCS'92,* 1992.

[309] Bernd-Olaf Kuppers. *Information and the Origin of Life.* The MIT Press, 1989. ISBN : 0-262-11142-X.

[310] Bradley Kuszmaul. The star tech massively parallel chess program. *Journal of the International Computer Chess Association,* 18(1), March 1995.

[311] Jules Lachelier. *Cours de logique.* Éditions Universitaires, 1990. Cours de logique donné (dans le cadre de l'enseignement de la philosophie) à l'École Normale Supérieure en 1866–1867.

[312] R.E. Ladner. The computational complexity of provability in systems of modal propositionnal logic. *SIAM J. Comput.,* 6(3) :467–480, 1977.

[313] Imre Lakatos. *Proofs and refutations : the logic of mathematical discovery.* Cambridge University Press, 1976. ISBN : 0-521-21078-X, 0-521-29038-4.

[314] Imre Lakatos. *Preuves et réfutations.* Hermann, 1984. Traduction de (Lakatos 1976).

[315] R. Lalement. *Logique, Réduction, Résolution.* Masson, Études et Recherches en informatique, 1990.

[316] P. Langley. Learning search strategies through discrimination. *International Journal on man-machine studies,* 18 :513–541, 1983.

[317] T. Langlois. *Algorithmes d'apprentissage par renforcement pour la commande adaptative.* Thèse de Doctorat de l'Universite de Technologie de Compiegne, 1992.

[318] Jean Largeault. *Principes classiques d'interprétation de la nature.* Librairie philosophique J. Vrin, 1988.

[319] J. Larrosa and P. Meseguer. Exploiting the use of DAC in max-CSP. In *Proc. of CP'96,* pages 308–322, Boston (MA), 1996.

[320] J. Larrosa and P. Meseguer. Partial lazy forward checking for max-csp. In *Proc. of the 13th ECAI,* pages 229–233, Brighton, United Kingdom, 1998.

[321] J. Larrosa, P. Meseguer, T. Schiex, and G. Verfaillie. Reversible DAC and other improvements for solving max-CSP. In *Proc. of the National Conf. on Artificial Intelligence (AAAI'98),* Madison, WIN, July 1998. American Association for Artificial Intelligence, AAAI Press and the MIT Press.

[322] J. Larrosa, P. Meseguer, and T. Schiex. Maintaining reversible DAC for Max-CSP. *Artificial Intelligence,* 107(1) :149–163, January 1999.

[323] Jean-Louis Laurière. *Un langage et un programme pour énoncer et résoudre des problèmes combinatoires.* Thèse de doctorat d'état, Université Pierre et Marie Curie, Paris, May 1976. Cf. Artificial Intelligence 10(1), p 29-127.

[324] J. L. Laurière. *Intelligence Artificielle, Tome I : résolution de problèmes par l'homme et la machine.* Eyrolles, 1986.

[325] Douglas Lenat. *AM : An Artificial Intelligence approach to discovery in Mathematics as heuristic search.* PhD thesis, Stanford University, 1976. Also published as Stanford University AI memo AIM-286.

[326] Xavier Leroy and Pierre Weis. *Manuel de référence du langage CAML.* InterÉditions, 1993.

[327] Xavier Leroy. *The Objective Caml system, documentation and user's guide.* Inria, 1998.

[328] David Lesaint. Specific solution sets for Constraint Satisfaction Problems. In *Proc. of the 13th international Avignon workshop*, 1993.

[329] David Levy and Monty Newborn. *How Computers Play Chess.* Computer Science Press, 1991. ISBN : 0-7167-8239-1, 0-7167-8121-2.

[330] David Levy, David Broughton, and Mark Taylor. The SEX algorithm in computer chess. *ICCA journal*, 12(1), 1989.

[331] Ming Li and Paul M.B. Vitányi. Kolmogorov complexity and its applications. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A : Algorithms and Complexity, chapter 4, pages 187–254. Elsevier, 1990.

[332] M. L. Littman, T. L. Dean, and L. P. Kaelbling. On the complexity of solving markov decision problems. In *Proceedings XXX-95.* XXX, 1995.

[333] M.L. Littman. Markov games as a framework for multi-agent reinforcement learning. In *International Conference on Machine Learning*, volume 11, pages 157–163, 1994.

[334] M. L. Littman. Value-function reinforcement learning in markov games. *Journal of Cognitive Systems Research*, 2 :55–66, 2001.

[335] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2nd extended edition, 1987. ISBN : 0-387-18199-7.

[336] Lionel Lobjois and Michel Lemaître. Branch and bound algorithm selection by performance prediction. In *Proc. of AAAI-98*, pages 353–358, Madison, WI, July 1998.

[337] W. S. Lovejoy. A survey of algorithmic methods for partially observed markov decision processes. *Annals of Operations Research*, 28(1) :65–47, 1991.

[338] Allan Mackintosh. L'ordinateur de John Atanassof. *Pour la science*, 132, Octobre 1988.

[339] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8 :99–118, 1977.

[340] Sridhar Mahadevan and Jonathan Connell. Automatic programming of behavior-based robots using reinforcement learning. *Artificial Intelligence*, 55 :311–365, 1992.

[341] S. Mahadevan, N. Marchalleck, T. Das, and A. Gosavi. Self-improving factory simulation using continuous-time average-reward reinforcement learning. In *Proceedings of the Fourteenth International Conference on Machine Learning (ICML'97)*, Nashville, TN, 1997.

[342] Misha Mahowald and Carver Mead. Une rétine en silicium. *Pour la Science*, 165, Juillet 1991.

[343] T. Anthony Marsland and Jonathan Schaeffer. *Computers, Chess and Cognition.* Springer-Verlag, 1990. ISBN : 0-387-97415-6.

[344] T. A. Marsland. A short history of computer chess. In *Computers, Chess and Cognition.* Springer-Verlag, 1990. In (Marsland and Schaeffer 1990).

[345] A. Martelli and U. Montanari. Optimization decision trees through heuristically guided search. *Communications of the ACM*, 21(12), 1978.

[346] P. Martin-Lof. The definition of random sequences. *Inform and Control*, 9, 1966.

[347] R. A. McCallum. Overcoming incomplete perception with utile distinction memory. *Proceedings of the Tenth International Machine Learning conference*, 1993.

[348] R. A. McCallum. *Reinforcement Learning with Selective Perception and Hidden State.* PhD thesis, Department of Computer Science. University of Rochester, 1995.

[349] John McCarthy. Chess as the Drosophilia of AI. In *Computers, Chess and Cognition.* Springer-Verlag, 1990. In (Marsland and Schaeffer 1990).

[350] W. S. McCulloch and W. A. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of mathematics and biophysics*, 5, 1943.

[351] R. J. McEliece, E.C. Posner, E.R. Rodernich, and S.S Venkatesh. Capacity of the hopfield associative memory. *IEEE transaction on information theory*, 33 :461–482, 1987.

[352] Maurice Merleau-Ponty. *La phénoménologie de la perception.* Gallimard, 1945.

[353] P. Meseguer and J. Larrosa. Constraint satisfaction as global optimization. In *Proc. of the 14th IJCAI*, pages 579–584, Montréal, Canada, 1995.

[354] Pedro Meseguer and Toby Walsh. Interleaved and discrepancy based search. In *Proc. of the 13th ECAI*, Brighton, United Kingdom, 1998.

[355] Pedro Meseguer. Interleaved depth first search. In *Proc. of the 15th IJCAI*, Nagoya, Aichi, Japan, 1997.

[356] N. Meuleau and P. Bourgine. Exploration of multi-states environments : local measures and back propagation of uncertainty. *Machine Learning*, 1998. To appear.

[357] N. Meuleau. *Le Dilemme entre Exploration et Exploitation dans l'Apprentissage par Renforcement.* Cemagref, Thèse de Doctorat de l'Universite de Caen, 1996.

[358] Zbigniew Michalewicz. *Genetic algorithms+data structures=evolution programs.* Springer-Verlag, 1992. ISBN : 0-387-55387-.

[359] Donald Michie. Brute force in Chess and Science. In *Computers, Chess and Cognition.* Springer-Verlag, 1990. In (Marsland and Schaeffer 1990).

[360] R. Milner, Tofte, and Harper. *Definition of Standard ML.* MIT Press, 1990.

[361] Marvin Minsky and Seymour Papert. *Perceptrons : an introduction to computational geometry.* MIT Press, 1969. ISBN : 0-262-63111-3.

[362] Marvin Minsky. *Computation : finite and infinite machines.* Prentice-Hall, 1967. LC Card Number : 67012342.

[363] Marvin Minsky. A framework for representating knowledge. In Patrick Winston, editor, *The Psychology of Computer Vision.* McGraw-Hill, 1975. ISBN : 0-07-071048-1.

[364] S. Minton, M. D. Johnston, A. B. Philips, and P. Laird. Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In *Proc. of AAAI-90*, pages 17–24, Boston, MA, 1990.

[365] Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. Minimizing conflicts : a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58 :160–205, December 1992.

[366] T. M. Mitchell. Learning and problem solving. In *Proceedings of IJCAI-83*, pages 1139–1151, 1983.

[367] Sanjay Mittal and Bernard A. Nadel. Constraint based reasoning — theory and applications. In *Tutorial printouts — 7th IEEE Conference on Artificial Intelligence Applications*, 1990.

[368] R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28(2) :225–233, 1986.

[369] R. Mohr and G. Masini. Good old discrete relaxation. In *Proc. of the 8th ECAI*, pages 651–656, Munchen FRG, 1988.

[370] U. Montanari. Network of constraints : Fundamental properties and applications to picture processing. *Inf. Sci.*, 7 :95–132, 1974.

[371] T. N. Buiand B. R. Moon. Analysing hyperplane synthesis in genetic algorithms using clustered schemata. Technical Report Cse-94-026, Dep. of Comp. Sci. and Engineering, Penn. State University, March 1994.

[372] A.W. Moore and C.G. Atkeson. Prioritized sweeping : Reinforcement learning with less data and less real time. *Machine Learning*, 13, 1993.

[373] A.W. Moore. The Parti-game Algorithm for Variable Resolution Reinforcement Learning in Multidimensional State-spaces. *NIPS'93*, 1993.

[374] R. Munos. A convergent Reinforcement Learning algorithm in the contunious case : the Finite-Element Reinforcement Learning. In *Proceedings of the Thirteenth International Conference on Machine Learning (ICML'96)*, Bari, Italy, July 1996.

[375] Bernard A. Nadel. Tree search and arc consistency in constraint satisfaction algorithms. In L. Kanal and V. Kumar, editors, *Search in Artificial Intelligence*, chapter 9, pages 287–342. Springer-Verlag, 1988. author also none as Nudel.

[376] Hiroshi Nakashima, Katsuto Nakajima, Seiichi Kondo, Yasutaka Takeda, Yu Inamura, Satoshi Onishi, and Kanae Masuda. Architecture and implementation of PIM/m. In *Proceedings of FGCS'92*, 1992.

[377] S. Ndiaye. *Apprentissage par renforcement en horizon fini : application à la génération de règles pour la Conduite de Culture.* Thèse de Doctorat de l'Universite Paul Sabatier, Toulouse, février 1999.

[378] R. M. Needham. Using cryptography for authentication. In Sape Mullander, editor, *Distributed systems.* ACM Press, 1990. ISBN : 0-201-41660-3.

[379] A. Newell, H. Simon, and J. Shaw. Preliminary report of general problem solving. Working paper 7, Carnegie Institute of Technology, 1957.

[380] Allen Newell. Physical symbol systems. In D. A. Norman, editor, *Perspectives on Cognitive Science*, pages 37–85. Norwood, NJ Ablex Publishing Corp., 1981. ISBN : 0-89391-071-6, ISBN : 0-89859-106-6.

[381] N. J. Nilsson. *Principles of artificial intelligence.* Morgan Kaufman, 1986. Reprint. ISBN : 0-934613-10-9.

[382] N. J. Nilsson. *Principes d'intelligence artificielle.* Cepadues, Toulouse, France, 1988. Edition originale : (Nilsson 1986).

[383] Maurice Nivat. *Cours d'Informatique fondamentale.* École Polytechnique, 1984.

[384] A. E. Nix and M. D. Vose. Modeling genetic algorithms with markov chains. In *Annals of Mathematics and Artificial Intelligence*, volume 5 (1), pages 79–88. Baltzer, 1992.

[385] Bernard A. Nudel. Consistent-labeling problems and their algorithms : Expected complexities and theory-based heuristics. *Artificial Intelligence*, 21 :135–178, 1983. Author also known as Nadel.

[386] W. P. Nuijten. *Time and Resource Constrained Scheduling : A Constraint Satisfaction Approach.* PhD thesis, Eindhoven University of Technology, 1994.

[387] L. Oxusoff and A. Rauzy. *Évaluation sémantique en calcul propositionnel.* Thèse de doctorat, Faculté des sciences de Luminy, Marseille, France, 1989.

[388] Ludek Pachman. *Théorie élémentaire du jeu d'échecs.* Payot, 1980.

[389] Olivier Palmade. *Étude et implantation de nouveaux mécanismes d'inférence en logique propositionnelle. Application aux ATMS.* Thèse de doctorat, Institut de Recherche en Informatique de Toulouse, Université Paul Sabatier, Toulouse, September 1992.

[390] C. H. Papadimitriou and J. N. Tsitsiklis. The complexity of Markov chain decision processes. *Journal of Mathematics of Operations Research*, 12(3) :441–450, 1987.

[391] Christos M. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, 1994. ISBN 0-201-53082-1.

[392] Jan Paredis. Co-evolutionary constraint satisfaction. In Y. Davido, H.-P. Schwefel, and R. Manner, editors, *International Conference on Evolutionary Computation, Parallel Problem Solving from Nature III*, volume 866 of *Lecture note in computer sciences*. Springer-Verlag, 1994.

[393] Jan Paredis. Coevolutionary computation. *Artificial Life Journal*, 2(4), 1996. MIT Pres / Bradford books.

[394] Jan Paredis. Coevolutionary life-time learning. In H.M. Voigt, M. Ebeling, I. Rechenberg, and H.P. Schwefel, editors, *International Conference on Evolutionary Computation, Parallel Problem Solving from Nature IV*, volume 1141 of *Lecture note in computer sciences*. Springer-Verlag, 1996.

[395] P. Pastor. *Etude et application des méthodes d'apprentissage pour la navigation en environnement inconnu*. Thèse de Doctorat de l'École Nationale Supérieure de l'Aéronautique et de l'Espace, Toulouse, 1998.

[396] M.S. Paterson and M. N. Wegman. Linear unification. *JCSS*, 16 :158–167, 1978.

[397] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991. ISBN :0-521-39022-2 (hb), 0-521-42225-6 (pb).

[398] Judea Pearl. *Heuristics — Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Publishing Comp., 1985.

[399] Judea Pearl. *Heuristique - Stratégies de recherche intelligentes pour la résolution de problèmes par ordinateur*. Cepadues Éditions, 1990. Traduction française de (Pearl 1985).

[400] J. Peng and R.J. Williams. Efficient learning and planning within the dyna framework. *Adaptive Behavior*, 1(4) :437–454, 1993.

[401] J. Peng and R.J. Williams. Incremental Multi-Step Q-Learning. In *International Conference on Machine Learning*, volume 11, pages 226–232, 1994.

[402] Roger Penrose. *The Emperor's New Mind : Concerning Computers, Minds, and the Laws of Physics*. Oxford University Press, 1989. ISBN : 0-19-851973-7.

[403] Roger Penrose. *L'esprit, l'ordinateur et les lois de la physique*. InterÉditions, 1992. Version française de (Penrose 1989).

[404] Christian Percebois et al. Projet COALA, articles et publications, 1984-1986. Technical report, LSI, 1986.

[405] Christian Percebois et al. Simulation results of a multiprocessor Prolog architecture based on a distributed and/or graph. Technical Report 6, Greco-Programmation, 1987.

[406] L. M. Pereira, F. Pereira, and D. H. D. Warren. *User's guide to DEC-10 PROLOG*. DAI, University of Edinburgh, 1979.

[407] Mark Perlin. Arc consistency for factorable relations. *Artificial Intelligence*, 53 :329–342, 1992.

[408] Rózsa Péter. *Jeux avec l'infini : voyage à travers les mathématiques*. Éditions du seuil, collection Points Sciences, 1977. L'édition hongroise originale date de 1957. ISBN : 2-02-004568-0.

[409] Gadi Pinkas. Propositional non-monotonic reasoning and inconsistency in symetric neural networks. In *Proc. of the 12th IJCAI*, pages 525–530, Sidney, Australia, 1991.

[410] Aske Plaat, Jonathan Schaeffer, Arie DeBruin, and Wim Pijls. A minimax algorithm better than sss^*. *Artificial Intelligence*, 87(1–2) :255–293, 1996.

[411] Krzystof Pomian, editor. *Le problème du déterminisme.* Gallimard, 1990.

[412] Bibliothèque pour la science. *Le cerveau.* Payot, 1987. ISBN 2-902918-24-0.

[413] PrologIA. *PrologIII, Manuel de référence et d'utilisation*, March 1990.

[414] Projet CSPFlex., Gérard Bel, Eric Bensana, Pierre Berlandier, Philippe David, Hélène Fargier, Christine Gaspin, Khaled Ghedira, Philippe Janssen, Philippe Jégou, Tibor Kökeny, Jérôme Lang, David Lesaint, Roger Martin-Clouaire, Bertrand Neveu, Jean-Pierre Rellier, Thomas Schiex, Brigitte Trousse, Gérard Verfaillie, and Marie-Catherine Vilarem. Représentation et traitement pratique de la flexibilité dans les problèmes sous contraintes. In *Actes des journées nationales du PRC GDR Intelligence Artificielle*, Marseille, France, October 1992.

[415] Patrick Prosser, Chris Conway, and Claude Muller. A constraint maintenance system for the distributed allocation problem. *Intelligent Systems Engineering*, 1(1) :76–83, 1992.

[416] Patrick Prosser. Hybrid algorithms for the Constraint Satisfaction Problem. *Computational Intelligence*, 9(3) :268–299, August 1993.

[417] Gregory M. Provan. The computational complexity of multiple-context truth maintenance systems. In *Proc. of the 8th ECAI*, pages 522–527, Stockholm, 1990.

[418] Jean-François Puget. On the satisfiability of symmetrical constrained satisfaction problems. In *Proc. of ISMIS'93*, 1993.

[419] P.W. Purdom. Search rearrangement backtracking and polynomial average time. *Artificial Intelligence*, 21 :117–133, 1983.

[420] M. L. Puterman and M. C. Shin. Modified policy iteration algorithms for discounted Markov decision processes. *Management Science*, 24 :1127–1137, 1978.

[421] M. L. Puterman. *Markov decision processes : discrete stochastic dynamic programming.* Wiley-Interscience, New York, 1994.

[422] Zenon Pylyshyn. *Computation and Cognition : Toward a Foundation for Cognitive Science.* MIT Press, 1984. ISBN : 0-262-16098-6.

[423] William J. Rapaport. Cognitive science. In A. Ralston and E. D. Reilly, editors, *Encyclopedia of computer science and engineering.* Van Rostrand Reinhold, 1983. ISBN : 0-442-24496-7.

[424] Thomas Ray. An approach to the synthesis of life. Technical report, School of Life and Science, University of Delaware, 1991.

[425] J.C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proc. of AAAI-94*, pages 362–367, Seattle, WA, 1994.

[426] Jean-Charles Régin. *Développement d'outils algorithmiques pour l'Intelligence Artificielle – Application à la chimie organique.* PhD thesis, Univeristé Montpellier III, December 1995.

[427] J.C. Régin. Generalized arc consistency for global cardinality constraints. In *Proc. of AAAI-96*, pages 362–367, Portland, OR, 1996.

[428] R. Reiter. On reasoning by defaults. In Ronald J. Brachman and Hector J. Levesque, editors, *Readings in knowledge representation.* Morgan Kaufman, 1987. ISBN : 0-934613-01-X (pbk.).

[429] N. Rescher and A. Urquhart. *Temporal logic.* Springer-Verlag, 1971. ISBN : 0-387-80995-3.

[430] Elaine Rich and Kevin Knight. *Artificial Intelligence.* McGraw-Hill, 1991. ISBN : 0-07-052263-4.

[431] Elaine Rich. *Artificial intelligence.* McGraw-Hill, 1983. ISBN : 0-07-052261-8.

[432] Elaine Rich. *Intelligence Artificielle.* Masson, 1987. Très bon livre de présentation de l'IA. Traduction de (Rich 1983). Une version plus récente existe en anglais : (Rich and Knight 1991).

[433] Jean-François Richard. *Les activités mentales. Comprendre, Raisonner, Trouver des solutions.* Armand Colin, 1990.

[434] G. D. Ritchie and F. K. Hanna. AM : A case study in AI methodology. *Artificial Intelligence*, 23 :249–268, 1984.

[435] François Rivenc. *Introduction à la logique.* Payot, 1989. ISBN : 2-228-88204-6.

[436] J. A. Robinson. A machine oriented logic based on the resolution principle. *JACM*, 1965.

[437] Dorothy Elizabeth Robling-Denning. *Cryptography and data security.* Addison-Wesley, 1982. ISBN : 0-201-10150-5.

[438] Irvin Rock and Stephen Palmer. L'héritage du Gestaltisme. *Pour la science*, 160, Février 1991.

[439] Roseaux. *Exemples et problèmes résolus de Recherche Opérationnelle.* Masson, 2nd edition, 1986.

[440] F. Rosenblatt. The perceptron : a probabilistic model for information storage and organization in the brain. *Psychoanalysis review*, 65, 1958.

[441] Fred Rosenblatt. *Principles of neurodynamics.* Spartan Books, 1962. LC Card Number : 62012882 /L/r85.

[442] A. Rosenfeld, R. Hummel, and S. Zucker. Scene labeling by relaxation operations. *IEEE Trans. on Systems, Man, and Cybernetics*, 6(6) :173–184, 1976.

[443] Francesca Rossi, Charles Petri, and Vasant Dhar. On the equivalence of constraint satisfaction problems. In *Proc. of the 9th European Conference on Artificial Intelligence*, pages 550–556, 1990.

[444] David E. Rumelhart. *Parallel Distributed Processsing : Explorations in the Microstructure of cognition, Volume 1 : Foundation*, chapter Learning internal representations by error propagation. MIT Press, 1986. ISBN : 0-262-18120-7.

[445] G.A. Rummery and M. Niranjan. On-line q-learning using connectionist systems. Technical report, Cambridge University Engineering Department, Cambridge, England, 1994.

[446] Bertrand Russell. *Histoire de mes idées philosophiques.* Gallimard, 1961.

[447] Bertrand Russell. *Écrits de logique philosophique*, chapter « Les principes des mathématiques ». PUF, 1989. Texte original : Principles of mathematics (1903).

[448] Bertrand Russell. *Écrits de logique philosophique*, chapter « La philosophie de l'atomisme logique ». PUF, 1989. Série de conférences prononcées par Russell en 1918.

[449] Daniel Sabin and Eugene C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of the 2nd International Workshop on Principles and Practice of Constraint Programming*, number 874 in LNCS, Rosario, Orcas Island (WA), May 1994. Springer-Verlag.

[450] Arto Salomaa. *Formal Languages.* Academic Press, New-York, 1973. LC Card Number : 72088356 //r84.

[451] Arto Salomaa. *Computation and automata.* Cambridge University Press, 1985. ISBN : 0-521-30245-5.

[452] Arto Salomaa. *Introduction à l'informatique théorique.* Armand Colin, 1989. Version française de (Salomaa 1985).

[453] Arthur Samuel. Some studies in machine learning using the game of checkers. *IBM journal of research development*, 3(3) :210–229, 1959.

[454] J.C. Santamaría, R.S. Sutton, and A. Ram. Experiments with reinforcement learning in problems with continuous state and action spaces. Technical Report 93-88, Georgia Institute of Technology and University of Massachusetts, 1996.

[455] V. Santos Costa, D. H. D. Warren, and R. Yang. The Andorra-I Engine : A parallel implementation of the Basic Andorra model. In *Logic Programming : Proceedings of the 8th International Conference*. MIT Press, 1991.

[456] V. Santos Costa, D. H. D. Warren, and R. Yang. The Andorra-I Preprocessor : Supporting full Prolog on the Basic Andorra model. In *Logic Programming : Proceedings of the 8th International Conference*. MIT Press, 1991.

[457] Vítor Santos Costa, David H. D. Warren, and Rong Yang. Andorra-I : A parallel Prolog system that transparently exploits both and- and or-parallelism. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM Press, April 1991.

[458] U. K. Sarkar, P. P. Chakrabarti, S. Ghose, and S. C. De Sarkar. Reducing reexpansions in iterative deepening search by controlling cutoff bounds. *Artificial Intelligence*, 50 :207–221, 1991.

[459] S. Schaal and C. Atkeson. Robot juggling : An implementation of memory-based learning. *Control System Magazine*, 1994.

[460] J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron. Reviving the game of checkers. In D. N. L. Levy and D. F. Beal, editors, *Heuristic Programming in Artificial Intelligence, the second Olympiad*, pages 161–173. Ellis Horwood, London, 1991. ISBN : 0-13-382615-5.

[461] Jonathan Schaeffer, Joseph Culbertson, Norman Treloar, Brent Knight, Paul Lu, and Duane Szafron. A world championship caliber checkers program. *Artificial Intelligence*, 1992.

[462] Jonathan Schaeffer. Distributed game tree searching. *Journal of parallel and distributed computing*, 6(2) :90–114, 1989.

[463] Jonathan Schaeffer. The history heuristics and alpha-beta search enhancemants in practice. *IEEE transactions on pattern analysis and machine intelligence*, 11(11) :1203–1212, 1989.

[464] Jonathan Schaeffer. Checkers : a preview of what will happen in chess ? *Journal of the International Computer Chess Association*, 14(2) :71–78, 1991.

[465] R. C. Schank. *Conceptual information processing*. North Holland, 1975. ISBN : 0-444-10773-8.

[466] Thomas Schiex and Gérard Verfaillie. No-good recording for static and dynamic CSP. In *Proceeding of the 5th IEEE International Conference on Tools with Artificial Intelligence (ICTAI93)*, pages 48–55, Boston, MA, November 1993.

[467] Thomas Schiex and Gérard Verfaillie. Two approaches to the solution maintenance problem in dynamic constraint satisfaction problems. In *Proc. of the IJCAI-93/SIGMAN Workshop on Knowledge-based Production Planning, Scheduling and Control*, pages 305–316, Chambery, France, August 1993.

[468] Thomas Schiex and Marie-Catherine Vilarem. Tutoriel sur les problèmes de satisfaction de contraintes et les outils existants. Journées nationales du PRC-IA., October 1992.

[469] T. Schiex, H. Fargier, and G. Verfaillie. Valued constraint satisfaction problems : hard and easy problems. In *Proc. of the 14th IJCAI*, pages 631–637, Montréal, Canada, August 1995.

[470] Thomas Schiex, Jean-Charles. Régin, Chistine Gaspin, and Gérard Verfaillie. Lazy arc consistency. In *Proc. of AAAI-96*, Portland, OR, August 1996. AAAI Press.

[471] T. Schiex. Possibilistic constraint satisfaction problems or "How to handle soft constraints ?". In *Proc. of the 8th Int. Conf. on Uncertainty in Artificial Intelligence*, Stanford, CA, July 1992.

[472] T. Schiex. Arc consistency for soft constraints. In *Actes de CP'00*, Singapour, September 2000.

[473] Marc Schoenauer, Edmund Ronald, and Sylvain Damour. Evolving nets for control. Technical report, CMAPX, Ecole Polytechnique, 91128 Palaiseau, France, May 1993.

[474] Nicol N. Schraudolf and John J. Grefenstette. *A user's guide to GAucsd 1.2.* UCSD, 1991.

[475] N. Schraudolph, P. Dayan, and T. Sejnowski. Temporal Difference Learning of Position Evaluation in the Game of Go. In J. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems 6.* Morgan Kaufmann, 1994.

[476] John R. Searle. *Du cerveau au savoir.* Hermann, 1985. Ensemble de conférences prononcées par Searle pour la BBC. Un livre remarquable par sa clarté et sa concision. Édition originale : 1984.

[477] John Searle. Minds, brains and programs. In Rainer Born, editor, *Artificial Intelligence, the case against.* Croom Helm, 1987. ISBN : 0-312-00439-7.

[478] John Searle. L'esprit est-il un programme d'ordinateur ? *Pour la science*, 149, Mars 1990.

[479] R. Seidel. A new method for solving constraint satisfaction problems. In *Proc. of the 7th IJCAI*, pages 338–342, Vancouver, Canada, 1981.

[480] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In *Proc. of AAAI-92*, pages 440–446, San Jose, CA, 1992.

[481] Adi Shamir. A polynomial time algorithm for breaking the basic Merkle-Hellman problem. *Proceedings of the 23rd FOCS Symposium*, 1982.

[482] Benny Shanon. Le Ménon, une conception de psychologie cognitive. In Monique Canto-Sperber, editor, *Les paradoxes de la connaissance.* Édition Odile Jacob, 1991.

[483] L. Shapiro and R. Haralick. Structural descriptions and inexact matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 3 :504–519, 1981.

[484] K. Shirayanagi. Knowledge representation and its refinement in Go programs. In *Computers, Chess and Cognition.* Springer-Verlag, 1990. In (Marsland and Schaeffer 1990).

[485] G. Sidebottom and W. S. Havens. Hierarchical arc consistency applied to numeric processing in constraint logic programming. Technical report, Center for Systems Science, Simon Fraser University, Burnaby, Canada, 1991.

[486] Pierre Siegel. *Représentation et utilisation de la connaissance en calcul propositionnel.* Thèse de doctorat d'état, Groupe Intelligence Artificielle, Université d'Aix Marseille II, Marseille, France, July 1987.

[487] H. Simon and J. B. Kadane. Optimal problem-solving search : all-or-none solutions. *Artificial intelligence*, 6(3), 1975.

[488] H. A. Simon. *The sciences of the artificial.* MIT Press, 2d ed., rev. and enl. edition, 1981. ISBN : 0-262-19193-8, ISBN : 0-262-69073-X (pbk.).

[489] Herbert Alexander Simon. *Sciences des systèmes – Sciences de l'artificiel.* AFCET Systèmes – DUNOT, 1991. ISBN : 2-04-019815-6, Traduction (Simon 1981).

[490] S. P. Singh and P. Dayan. Analytical Mean Squared Error Curves for Temporal Difference learning. *Machine Learning*, 32(1) :5–40, 1998.

[491] S. P. Singh and R. S. Sutton. Reinforcement learning with replacing eligibility traces. *Machine Learning*, 1996.

[492] S.P. Singh, T. Jaakkola, and M.I. Jordan. Learning Without State-Estimation in Partially Observable Markovian Decision Processes. *Machine Learning (ML'94)*, pages 284–292, 1994.

[493] M. Singh. Path consistency revisited. In *ICTAI'95 (IEEE Conference of Tools for Artificial Intelligence)*, Washington D.C., 1995.

[494] Françoise Siri. von neumann et la génèse des sciences cognitives. *La recherche*, 252, March 1993.

[495] D. J. Slate and L. R. Atkin. Chess 4.5, the northwestern university chess program. In P. Frey, editor, *Chess Skill in Man and Machine*, pages 82–118. Springer-Verlag, 1978. ISBN : 0-387-07957-2.

[496] Derek Smith and Paul Stanford. A random walk in hamming space. In *Proceedings of the International Joint Conference on Neural Networks*, volume 2, 1990.

[497] R. E. Smith, D. E. Goldberg, and J. A. Earickson. *SGA-C : A C-language implementation of a Simple Genetic Algorithm*, May 1991. TCGA report No. 91002.

[498] Raymond Smullyan. *Quel est le titre de ce livre ?* Dunod, 1981.

[499] Françoise Fogelman Soulié. Learning in automate networks. In *Artificial intelligence*. CIIAM, 1986.

[500] Oswald Spengler. *Le déclin de l'Occident, Tome I : Forme et réalité.* Gallimard, 1948.

[501] R.M. Stallman and G.J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9(2) :135–196, 1977.

[502] L Sterling and E. Shapiro. *The Art of Prolog : advanced programming techniques.* The MIT Press, USA, 1986. ISBN : 0-262-19250-0, ISBN : 0-262-69105-1 (pbk.).

[503] R.S. Sutton and A.G. Barto. *Reinforcement Learning : An Introduction.* MIT Press, Cambridge, Massachusetts, 1998.

[504] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. *NIPS-12*, 1999.

[505] R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3 :9–44, 1988.

[506] R.S. Sutton. Integrated Architectures for Learning, Planning and Reacting Based on Approximating Dynamic Programming. In *International Conference on Machine Learning*, volume 7, pages 216–224, 1990.

[507] R.S. Sutton. Generalization in Reinforcement Learning : Successful Examples using Sparse Coarse Coding. *NIPS'95*, 1995.

[508] Kazuo Taki. Parallel Inference Machine PIM. In *Proceedings of FGCS'92*, 1992.

[509] Hidetoshi Tanaka. Integrated system for protein information processing. In *Proceedings of FGCS'92*, 1992.

[510] R. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13(3) :566–579, 1984.

[511] G. J. Tesauro. Practical issues in temporal difference learning. In John E. Moody, Steven J. Hanson, and Richard P. Lippmann, editors, *Advances in Neural Information Processing Systems 4*, pages 260–266. Morgan Kaufmann, 1992.

[512] G. J. Tesauro. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2) :215–219, 1994.

[513] Rudolf Von Thadden. Télérama, Mercredi 1er Janvier 1991.

[514] André Thaysse et al. *Approche logique de l'intelligence artificielle, Tome I : de la logique classique à la programmation logique.* Dunod, 1988.

[515] S.B. Thrun. The role of exploration in learning control. *HandBook of Intelligent Control : Neural, Fuzzy and Adaptive Approaches*, 1992.

[516] Guy Tiberghien. *Psychologie cognitive : modèles et méthodes*, chapter Introduction : modèles de l'activité cognitive. Presses Universitaires de Grenoble, 1988.

[517] TMC. Connection machine – Model CM5 – Technical summary. Technical report, Thinking Machine Company, Cambridge, Massachusetts, October 1991.

[518] J. N. Tsitsiklis and B. Van Roy. Feature-based methods for large scale dynamic programming. *Machine Learning*, 22 :59–94, 1996.

[519] Alan Mathison Turing. Computing machinery and intelligence,. In Alan Ross Anderson, editor, *Minds and Machines*. Englewoods Cliffs, 1964.

[520] Alan Mathison Turing. *Pensée et machine*, chapter « Les ordinateurs et l'intelligence ». Éditions du champ Vallon, 1983. Traduction de (Turing 1964).

[521] David Turner. An overview of miranda. *ACM SIGPLAN Notices*, 21 :156–166, 1986.

[522] Raymond Turner. *Logiques pour l'Intelligence Artificielle*. Masson, 1986.

[523] J. R. Ullman. Associating parts of patterns. *Inform. Contr.*, 9 :583–601, 1966.

[524] John Marshall Unger. *The Fifth Generation Fallacy : Why Japan is Betting Its Future on Artificial Intelligence*. Oxford University Press, 1987. ISBN : 0-19-504939-X.

[525] W. Uther and M. Veloso. Tree Based Discretization for Continuous State Space Reinforcement Learning. In *AAAI National Conference*, volume 15, pages 769–774, 1998.

[526] Peter van Beek and Rina Dechter. Constraint tightness and looseness versus local and global consistency. *JACM*, 1997.

[527] P. van Beek. Approximation algorithms for temporal reasoning. In *Proc. of the 11th IJCAI*, pages 1291–1296, Detroit, MI, 1989.

[528] Peter van Beek. On the minimality and decomposability of constraint networks. In *Proc. of AAAI-92*, pages 447–452, San Jose, CA, 1992.

[529] Peter van Beek. Reasoning about qualitative temporal information. *Artificial Intelligence*, 58 :297–326, December 1992.

[530] Michel van Caneghem. *L'anatomie de PROLOG*. InterÉditions, 1986.

[531] Peter van Emde Boas. Machine models and simulations. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A : Algorithms and Complexity, chapter 1, pages 1–66. Elsevier, 1990.

[532] Pascal van Hentenryck and Yves Deville. The cardinality operator : A new logical connective for constraint logic programming. In *Proc. of the 8th international conference on logic programming*, Paris, France, June 1991.

[533] P. van Hentenryck and T. Le Provost. Incremental search in contraint logic programming. *New Generation Computing*, 9 :257–275, 1991.

[534] P. van Hentenryck, Y. Deville, and C. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57 :291–321, 1992.

[535] Francisco Varela, Evan Thompson, and Eleanor Rosch. *The Embodied Mind : Cognitive Science and Human Experience*. The MIT Press, Cambridge, MA, 1991. ISBN : 0-262-22042-3.

[536] Nageshwara Rao Vempaty. Solving constraint satisfaction problems using finite state automata. In *Proc. of AAAI-92*, pages 453–458, San Jose, CA, 1992.

[537] Peter ven Emde Boas. Machines models and simulations. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A : Algorithms and Complexity, chapter 1, pages 1–66. Elsevier, 1990.

[538] G. Verfaillie, M. Lemaître, and T. Schiex. Russian doll search. In *Proc. of AAAI-96*, pages 181–187, Portland, OR, 1996.

[539] Gérard Verfaillie. Problèmes de satisfaction de contraintes : production et révision de solution par modifications locales. In *Proc. of the 13th international Avignon workshop*, pages 277–286, 1993.

[540] Kim J. Vicente and William F. Brewer. La mémoire trompeuse des scientifiques. *La recherche*, 24(258), October 1993.

[541] M.C. Vilarem, P. Janssen, C. Bessière, and P. David. Satisfaction de contraintes : principes et algorithmique. In *Actes de l'école d'été du PRC-IA*, Marseille, France, September 1991.

[542] John von Neumann. *L'ordinateur et le cerveau.* La découverte, 1992. Traduction de Pascal Engel.

[543] M. D. Vose. Generalizing the notion of schema in genetic algorithms. *Artificial Intelligence*, 50 :385–396, 1991.

[544] R. Wallace. Directed arc consistency preprocessing. In M. Meyer, editor, *Selected papers from the ECAI-94 Workshop on Constraint Processing*, number 923 in LNCS, pages 121–137. Springer, Berlin, 1995.

[545] Toby Walsh. Depth-bounded discrepancy search. In *Proc. of the 15th IJCAI*, pages 1388–1393, Nagoya, Aichi, Japan, 1997.

[546] D. L. Waltz. Generating semantic descriptions from drawings of scenes with shadows. Technical Report AI271, M.I.T., Cambridge MA, 1972.

[547] Hao Wang. *Kurt Gödel.* Armand Colin, 1990.

[548] D. H. D. Warren. Implementing PROLOG : Compiling predicate logic programs. report 39-40, DAI, 1977.

[549] D. H. D. Warren. An abstract PROLOG instruction set. note 309, SRI international, October 1983.

[550] C.J.C.H. Watkins. *Learning from Delayed Rewards.* PhD thesis, Cambridge University, Cambridge, England, 1989.

[551] Jean-Christophe Weill. *Programme d'échecs de championnat, architecture logicielle, synthèse de fonction d'évaluation, parallélisme de recherche.* PhD thesis, Université paris 8, Janvier 1995.

[552] Pierre Weis and Xavier Leroy. *Le langage CAML.* InterÉditions, 1993.

[553] Pierre Weis and Xavier Leroy. *Le langage Caml.* InterEditions, 1993.

[554] Joseph Weizenbaum. *Computer Power and Human Reason.* Penguin, 1984. 2nd edition. ISBN : 0-14-022535-8 (pbk.).

[555] S.D. Whitehead and L.-J. Lin. Reinforcement Learning of non-Markov decision processes. *Artificial Intelligence*, 73 :271–306, 1995.

[556] M. A. Wiering and J. Schmidhuber. Fast online Q(λ). *Machine Learning*, 33(1) :105–115, 1998.

[557] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8 :229–256, 1992.

[558] Terry Winograd and Fernando Flores. *L'intelligence artificielle en question.* InterÉditions, 1986. Traduction de (Winograd and Flores 1986b).

[559] Terry Winograd and Fernando Flores. *Understanding computers and cognition : a new foundation for design.* Ablex Pub. Corp., 1986. ISBN : 0-89391-050-3.

[560] Patrick Winston. *Artificial Intelligence.* Addison-Wesley, 1984.

[561] Ludwig Wittgenstein. *Investigations philosophiques.* Gallimard, 1961. Édition originale : Janvier 1945.

[562] Ludwig Wittgenstein. *Tractatus Logico-philosophicus*. Gallimard, 1961. Édition originale : 1918.

[563] Pierre Wolper. *Introduction à la calculabilité*. InterÉditions, 1991.

[564] Nguyen Huy Xuong. *Mathématiques discrètes et informatique*. Masson (Logique Mathématique Informatique), Paris, France, 1992. ISBN : 2-225-82621-8.

[565] Hiroshi Yashiro, Tetsuro Fujise, Takashi Chikayama, Masahiro Matsuo, Atsushi Hori, and Kumiko Wada. Resource management mechanism of PIMOS. In *Proceedings of FGCS'92*, 1992.

[566] Hideki Yasukawa, Hiroshi Tsuda, and Kazumasa Yokota. Objects, properties and modules in QUIXOTE. In *Proceedings of FGCS'92*, 1992.

[567] X. Yin and N. Germay. A fast genetic algorithm with sharing scheme using cluster analysis methods in multimodal function optimization. In *Proceedings of the Artificial Neural Nets and Genetic Algorithms Conference*, 1993.

[568] Kaoru Yoshida, Cassandra Smith, Toni Kazic, Georges Michaels, Ron Taylor, David Zawada, Ray Hagstrom, and Ross Overbeek. Toward a human genome encyclopedia. In *Proceedings of FGCS'92*, 1992.

[569] Ramin Zabih. Some applications of graph bandwidth to constraint satisfaction problems. In *Proc. of AAAI-90*, pages 46–51, Boston, MA, 1990.

[570] L. A. Zadeh. Fuzzy sets. *Inform and Control*, 8, 1965.

[571] L. A. Zadeh. Fuzzy logic and its applications to approximate reasonning. In *Information Processing 74, IFIP Congress, Amsterdam*, 1974.

[572] W. Zhang and T. Dietterich. A reinforcement learning approach to job-shop scheduling. In *IJCAI' 95*, 1995.

[573] Anatoly A. Zhigljavsky. *Theory of Global Random Search*. Kluwer Academic Plubishers, 1991.

Index

Symboles

α-β ..... *voir* algorithmes
α-équivalence 178
β-contraction 179
β-équivalence 183
β-redex 179
  maximal 182
β-réduction 179
$\beta\eta$-redex 184
$\beta\eta$-réduction 184
Δ-voyageur de commerce 168
ε-approximation 166
η-redex 184
η-réduction 183
$\exists$ 56
$\forall$ 56
λ-calcul 65, 175–202
  α-équivalence 178
  β-contraction 179
  β-équivalence 183
    indécidabilité 183
  β-redex 179
    maximal 182
  β-réduction 179
  $\beta\eta$-redex 184
  $\beta\eta$-réduction 184
  η-redex 184
  η-réduction 183
  équations 186
  abstractions 176
  applications 176
  appliqué 176, 191
  combinateurs 177
    de point fixe 185
    K 179
    S 179
    Y 185
    Y_{Curry} 185
    Y_{Turing} 186
    Y_{Klop} 185
  confluence 181
  contractum 179
  conventions simplificatrices 176
  curryfication 177
  équivalence syntaxique 176
  extensionnalité 183
  leftmost outermost reduction 181
  nombres de Church 189
  opérateur de point fixe 184
  propriété du diamant 181
  pur 176
  récursivité 184
  représentation des booléens 187
  représentation des doublets 188
  représentation des entiers 189
  simplement typé 192
    décidabilité 195
    nœthérien 194
    termes 193
    types 192
    types atomiques 192
  stratégie de réduction 181
  substitution 178
  termes 176
    clos ..... *voir* combinateurs
    forme normale 180
  théorème de Church-Rosser 181
  typé 192–202
  typage inféré 195
    à la Damas-Milner 199
    contexte de typage 196, 200
    décidabilité 198
    `let` 199
    polymorphique 199
    schémas de type 199
    substitution de type 196
    système de typage 196
    terme stratifié 198
    type principal 198
    types avec variables 195
    variable générique 200
  variables 176
    liées 177
    libres 177
    occurrences liées 177
    occurrences libres 177
    portée 177
    renommages 178
2-SAT 163
3-SAT 163

A

A* ..... *voir* algorithmes
abstraction 176
adéquation 51, 63
affectation de type 195
agendas 333
algorithmes 67
  α-β 270, 271, 300
    avec mémoire 282
  A* 218, 358
    complexité 220
  A^*_ε 221
  AO* 221
  British Museum 213
  complexité 153
  Davis et Putnam 97
  de Markov 137

langage reconnu 138
de programmation dynamique 450
Dijkstra 221
efficacité 153
escalade 216
filtrage d'un CSP 238
génétiques 415–441
codage 418
croisement 416, 421
exemple complet 422
instance d'un schéma 419
mutation 416, 421
ordre d'un schéma 420
reproduction 416, 420
schéma 419
iterative deepening 278
IterSSS* 276
largeur d'abord 215
meilleur d'abord 218
minimax 269
MTD(*f*) 283
parallélisation 284
profondeur d'abord 214
propagation de contraintes 238
Quine 94
satisfaction de contraintes ..... *voir* CSP
Scout 276
simplexe 228
SSS* 272
alphabet 66, 131, 132
AM 338
analogie 294
analyse rétrograde 298
antitautologie 46
AO* ..... *voir* algorithmes
appel par besoin 181, 349
appel par nom 181, 349
appel par valeur 181, 349
application 176
apprentissage 28, 245, 250, 288
de base 289
non-supervisé 444
par renforcement *voir* apprentissage par renforcement
supervisé 400, 401, 444
symbolique 389–393
déductif 391
empirique 391
généralisation 392
inductif 391
inventif 391
par explications 390, 391
par similarités 390, 392
apprentissage par renforcement 443–475
algorithmes 453–464
à base de modèle 454
méthodes de gradient 466, 473
Monte Carlo 459
Q-learning 455
Q(λ) 463
SARSA 463
TD(λ) 458
autonomie 443
essais-erreurs 444
exploration 456
historique 445, 453
méthodes de généralisation 464–473
approximations linéaires 467
arbre de régression 471
CMAC 468
fonctions de voisinage 467
fonctions heuristiques 469
partitionnement 468
perceptron multi-couches 470
règles de décision 471
radial basis functions 469
représentations différentiables 466
représentations paramétrées 465
observabilité partielle 472
processus décisionnel de Markov 445
professeur 444
récompenses retardées 444
simulation 453
traces d'éligibilité 461
arc-consistance 238
ARGOSL 338
arithmétique formelle 80
axiomes 81
axiomes non-logiques 81
consistance 83, 89
formules 81
incomplétude 87
indécidabilité 87
règles d'inférence 81
symboles 81
terme 81
ASA ..... *voir* apprentissage, symbolique
ATMS 335–337
complexité 337
contexte 336
environnement 336
hypothèses 336
justifications 336
label 336
nogood 336
atomes 42, 56
de base 108
autonomie 443
Awale 277
axiomatique 48
formelle 75
matérielle 75
axiomes 48, 149
axone 395

B

Babbage 267
backgammon 307, 474
backjumping 249
backtrack 236, 368
intelligent 100, 245, 249
base de faits 333, 364
base de Herbrand 108
base de règles 333, 364
Bayes 306
BILL 288, 474
bridge 304–307

C

calcul d'acceptation 68
calcul des prédicats 55, 322
calculabilité 65, 84, 139–147, 222, 484
call by name ..... *voir* appel par nom
call by need ..... *voir* appel par nécessité
call by value ..... *voir* appel par valeur
Cantor 76
caractère aléatoire 147
casses-têtes 263
catégoricité 91
cavalier d'Euler 264
chaînage arrière 208, 333, 334
chaînage avant 208, 333, 334
chaînage mixte 209, 334
CHARME 379
checkers ..... *voir* jeux, checkers
chemin-consistance 243
CHINOOK 307
CHIP 379
Church 66
Cinquième génération 28
circuit absorbant 155
circuit hamiltonien 164
classe EXPTIME ..... *voir* problèmes, classe EXPTIME
classe NPSPACE ..... *voir* problèmes, classe NPSPACE
classe NP ..... *voir* problèmes, classe NP
classe PSPACE ..... *voir* problèmes, classe PSPACE
classe P ..... *voir* problèmes, classe P
classe co-NP ..... *voir* problèmes, classe co-NP
classe co-P ..... *voir* problèmes, classe co-P
clauses 95
  de Horn 103, 364
clique 259
clôture existentielle 58
clôture universelle 58
CLP($\mathbb{R}$) 379
CLP($\mathcal{X}$) 379
CMAC 468
codage de données 153
code à clé publique 170
cogniticien 487
cognitif 483
combinateurs 177
  de point fixe 185
compilateur 356
complémentaire d'un problème 152
complétude 51, 63
  au sens de Post 123
  faible 52
  forte 51
  syntaxique 52
complexité 151–172, 222
  aléatoire 148
  logique 149
  organisée 149
  spatiale 171
  temporelle 151–171
composantes connexes 155
compositionalité 42
concaténation 131
conclusion 48
confluence 181
conjecture d'Erdös 264
connaissances 26, 28
connecteurs 42
  dyadiques 44
conséquence valide 47
consistance d'un CSP 232
consistance formelle 80
consistance globale d'un CSP 234
consistance locale d'un CSP . *voir* CSP, consistances locales
consistance sémantique 46, 61
consistance syntaxique 51
consistency enforcing ..... *voir* CSP, filtrage
contexte de typage 196, 200
contingence 46
contrôle du temps 278
contractum 179
contrainte 229, 382
  induite 234
  satisfaction 231
  violation 231
convention négamax 271
Conway ..... *voir* jeux, Conway
Cook 158
coupe-cycle 258
cryptarithmétique 263
cryptographie 170
CSP 227–266
  apprentissage 250
  classes polynomiales 257–260
  cohérence 232
  consistance 232
  consistance globale 234
  consistance locale 237–244
    arc-consistance 238, 242
    chemin-consistance 243
    k-consistance 237
    k-consistance forte 238
  contrainte induite 234
  décomposition
    coupe-cycle 258
    cycle cutset 258
    regroupement en hyper-arbre 259
    tree-clustering 259
  définition 229
  décomposable 235
  équivalence 233
  évaluation de performances 245
  exemples 263
  filtrage 237–244
    AC-Inférence 241
    AC3 239
    AC4 239
    AC5 240
    AC6 240
    AC7 241
    arc-consistance 242
    check backward 242
    check forward 242
    chemin-consistance 243
    complexité 238
    full look future 242
    k-consistance 238
    partial look future 242
  générés aléatoirement 266
  graphe 229
  heuristiques 256
  hyper-graphe 229
  instanciation 231
    cohérente 232
    consistante 232
    globalement consistante 232
  largeur 257

mémorisation de contraintes 250
micro-structure 229, 257
minimalité 235
nogood recording 250
ordres 245
heuristiques 245
horizontal 248
vérification des contraintes 245
vertical 245
propagation de contraintes ..... *voir* filtrage
requêtes 235
résolution 235–237, 244–256
backtrack 236
backtrack intelligent 250
forward-checking 253
full look ahead 253
generate and test 236
MAC 253
maintien de l'arc-consistance 253
partial look ahead 253
real full look ahead 253
test and generate 236
solution 232
structure 229
support 238
taux de satisfiabilité 231
valeur viable 238
Curry-Howard 196
curryfication 177
cut 375
cycle eulérien 155
cycle hamiltonien 164
cycle-cutset 258

D

Damas-Milner 199
dames ..... *voir* jeux, dames
américaines ..... *voir* jeux, checkers
DARPA 29
DART 338
Davis et Putnam 97
décidabilité 84, 139–147, 195
déclenchement 334
décomposabilité d'un CSP 235
DEEP THOUGHT 30, 289
démonstration 48, 49
automatique 28, 206
DENDRAL 27, 338
dendrite 395
dépendance conceptuelle 331
dérivation 133
déterminisme 74
diagonale cantorienne 77
Digital Equipment Corporation 37
Diophante 144
diploïdie 433
discernement 303
distance de Hamming 397
domaine de Herbrand 108

E

EBL ..... *voir* apprentissage, symbolique
échecs ..... *voir* jeux, échecs
efficacité d'un algorithme 153
élagage de futilité 300
ELIZA 32
énoncés idéaux 80
enseignement 22, 23
ensemble diophantien 144
degré 144
dimension 144
polynôme 144
ensembles 76
dénombrables 76
équation diophantienne 144
équivalence de CSP 233
escalade 216
espace de Hamming 397
espaces d'états 205, 206
chaînage arrière 208
chaînage avant 208
chaînage mixte 209
état d'un problème 206
raisonnement déductif 208
représentation par arbres 212
représentation par graphes 212
représentation par graphes ET-OU 213
représentation par graphes OU 213
solution inductive 208
stratégie de contrôle 205
système commutatif 209
système de production 205, 208
système monotonique 209
système partiellement commutatif 209
variables d'état 205
essais-erreurs 444
état d'un problème 206
évaluation 269, 350
évaluation sémantique 98
experts 487
extensionnalité 183

F

factorisation 112
feedforward 400
fenêtre de recherche 283
filtrage ..... *voir* CSP, filtrage
finales 287, 295
finitiste 80
fonctions 65, 68, 175
calculables 68, 84
d'évaluation 269, 288, 292, 298, 299
d'activation 402
de transition 67
heuristiques ..... *voir* heuristiques
λ-calculables 65
propositionnelles 55
récursives 184
récursives générales 65
récursives 349
Turing-calculables 65
fonctions récursives 353
formalistes 80
forme conjonctive normale 96
pure 97

forme de Skolem 106
forme normale 105, 135, 180
de Chomsky 135
forme prénexe 105
forme standard 108
formules 42, 56
bien formées 42, 56
cohérentes 61
consistantes 61
contingentes 46
insatisfiables 46
satisfiables 46, 61
valides 46, 61
forward-checking 253
frames 332
attributs 332
slots 332

G

Garbage Collector 348
générer et tester 236
géométrie 75
Go ..... *voir* jeux,Go
Gödel 78
grammaires 133
algébriques 134
forme normale de Chomsky 135
équivalentes 133
forme normale 135
libres de contexte 134
forme normale de Chomsky 135
régulières 134
forme normale 135
symboles non terminaux 133
symboles terminaux 133
granularité 284
graphe 229
complet 229
graphe d'un CSP 229
graphes ET-OU ..... *voir* espace d'états, graphes
graphes OU ..... *voir* espace d'états, graphes
Groot, de 290

H

Hackenbush ..... *voir* jeux, Hackenbush
haploïdie 433
hash-coding 280
HEARSAY-III 338
heuristiques 36, 215, 269, 298, 307
admissibles 220
consistantes 220
minorantes 220
monotones 220
parfaites 220
presque parfaites 220
HITECH 289
hyper-graphe d'un CSP 229
hypothèse du continu 77

I

IA 34
cognitive 30, 483
connexionniste 32, 490
critiques 480
définitions 25, 35
faible 31, 481
forte 30, 481
historique 26
postulat mécaniste 27
pragmatique 31
problèmes typiques 35
symbolique 489
IBM 29
implication matérielle 45
incomplétude 148, 484
indécidabilité 139–147
infini 76
ingénierie cognitive 322
instance fondamentale 110
instance générique 200
instanciation 231
cohérente 232
consistante 232
globalement consistante 232
intelligence artificielle ..... *voir* IA
interprétation 44, 60
de Herbrand 109
intraitabilité ..... *voir* problèmes, intraitables
intuitionnistes 79
IPL1 27
itérateur 354
itération de la stratégie 451
itération de la valeur 451
IterSSS* 276

J

jeux 28, 267–319
Awale 277
backgammon 307
TDGammon 474
bridge 304–307
annonces 304
jeu de la carte 305
checkers 27, 267, 308, 474
CHINOOK 307
Chomp 318
Conway 310–319
comparaison de jeux 315
nimbers 317
nimbres 317
nombres 315
ppex 317
somme de jeux 313
Cross-Cram 318
dames 308
américaines ..... *voir* checkers
discernement 303
échecs 37, 207, 211, 267, 289–304, 480
finales 295
KnightCap 474
milieu de partie 299
ouvertures 294
psychologie 290–294

Go 309, 474
MANY FACES OF GO 309
Hackenbush 311
heuristiques ..... *voir* fonctions d'évaluation
impartiaux 315
Kalah 277
Othello 286–288
finales 287
milieu de partie 287
ouvertures 286
programmation ..... *voir* algorithmes
analyse rétrograde 298
apprentissage 288, 474
apprentissage de base 289
contrôle du temps 278
coup meurtrier 301
coup nul 303
élagage de futilité 300
fenêtre de recherche 283
finales 287, 295
milieu de partie 287, 299
ouvertures 286, 294
recherche secondaire 301
retour à l'équilibre 301
search extension 302
tables de transposition 279
psychologie 290–294
Reversi ..... *voir* Othello
Sprouts 319
stratégie 273, 300
partielle 273
tactique 300

K

k-consistance 237
forte 237
Kalah ..... *voir* jeux, Kalah
KnightCap ..... *voir* jeux,échecs

L

label 336
lambda-calcul ..... *voir* λ-calcul
langage naturel 28
langages à objets 332
langages formels 131–138
algébriques 134
alphabet 131, 132
concaténation 131
grammaires ..... *voir* grammaires
libres de contexte 134
morphisme 132
mots 131
récursif 133
récursivement énumérables 133
réguliers 134
symboles 131
système de reconnaissance 132
système génératif 132
systèmes de réécriture ..... *voir* systèmes de réécriture
largeur d'un CSP 257
leftmost outermost reduction 181
lemme de Shurr 264
Levy 480
LISP 27, 66, 337
listes 354
littéral 43
négatif 43
positif 43
logique absolue 115
logique classique 322
logique combinatoire 179, 196
logique des défauts 327
normale 328
semi-normale 329
logique des prédicats 55–63
adéquation 63
atome 56
de base 108
axiomes 58
base de Herbrand 108
calcul 105–114
complétude 63
consistance sémantique 61
déduction 59
domaine de Herbrand 108
forme clausale 108
forme de Skolem 106
forme normale 105
forme prénexe 105
forme standard 108
formules 56
atomiques 56
closes 57
cohérentes 61
consistantes 61
fermées 57
satisfiables 61
indécidabilité 63
instance d'un terme 110
instance d'une clause 110
interprétation 60
de Herbrand 109
littéraux unifiables 111
matrice 105
modèles 59, 61
de Herbrand 109
plus général unificateur 111
prédicats 55
principe de résolution 112
quantificateurs 56
règles d'inférence 59
relation de conséquence 62
résolution ..... *voir* calcul
sémantique 59
skolémisation 106
substitution 110
de renommage 111
pure 110
terme 56
de base 108
unificateur 111
unification 111
occur-check 112
test d'occurrence 112
univers de Herbrand 108
validité 61
variables 56
liées 56
libres 56
logique des propositions 41–53

adéquation 51
antitautologie 45
atome 42
axiomatique 41, 48
axiomes 48
calcul 93–104
arbres sémantiques 94
Davis et Putnam 97
évaluation sémantique 98
heuristiques 100
partition des modèles 99
principe de résolution 102
Quine 94
réduction 95
clauses 95
de Horn 103
cohérence ..... *voir* consistance
complétude 51
compositionalité 42
connecteurs 42
conséquence valide 47
consistance 51
décidabilité 52
déduction 50
démonstration 48, 49
forme conjonctive normale 96
forme conjonctive normale pure 97
forme normale 95
formules 42
implication matérielle 45
interprétation 44
littéral 43
modèles 43
principe de déduction 93
règles d'inférence 49
sémantique 41, 43
satisfiabilité 43, 44
substitution 46
tables de vérité 43, 44
tautologie 45
théorème 48
validité 45
valuation 44
variable 42
logique du premier ordre ..... *voir* logique des prédicats
logique floue 326
logique intuitionniste 117
logique minimale 117
logique modale 117, 323
adéquation 123
aléthique 119
complétude 123
complexité 172
consistance 123
épistémique 123
modèle de Kripke 121
modèle de S4 122
modèle de S5 122
modalités 118
multi-modale 124
sémantique 120
système S4 120
système S5 ..... 120, 324
système T 119
temporelle 124
logique positive 116
logique propositionnelle ..... *voir* logique des propositions
logique temporelle 124, 325
de Allen 325
logiques faibles 115
logiques multi-valuées 326
logiques non classiques 115–127
look ahead 253

M

machines de Turing 65–71
alphabet 66
calcul d'acceptation 68
configuration 67
déterministes 66
complexité temporelle 153
états 67
fonction de transition 67
index 85
langage accepté 68
non déterministes 70
complexité temporelle 153
séquence de calcul 67
MACSYMA 34, 340
maintien de cohérence 335
MANY FACES OF GO 309
many-one reduction 156
Mathématique 73
mémoire associative 396
mémorisation 291
mémorisation de contraintes 245, 250
META-DENDRAL 338
méta-règles 333
méthodes faibles 205
micro-mondes 268, 482
milieu de partie 287, 299
minimalité d'un CSP 235
minimax 268, 269, 474
modèles 43
canoniques 103
de Herbrand 109
modèle de Hopfield 398
modularité 360
monotonie 335
morphisme 132
moteur d'inférence 333, 334, 365
chaînage arrière 334
chaînage avant 334
chaînage mixte 334
déclenchement 334
monotonie 335
résolution de conflit 334
régime de contrôle 334
à tentative 334
irrévocable 334
sélection 334
mutation 416
MYCIN 37, 338, 340

N

n reines 264
négamax 268, 271
Newell 267, 480
nogood 336
nogood recording 245
nombres algébriques 76
nombres de Church 189
nombres de Conway 315
nombres de Grundy 317
nombres de Ramsey 266
normalisation 96
nœthérien 195

O

OCAML 347–361
portée lexicale 348
syntaxe 349
occur-check 112
opérateur de point fixe 184
optimisation 224, 416
ordonnancement 165
ordre de réduction standard 181
ordre supérieur 352
Othello ..... *voir* jeux, Othello
ouvertures à Othello 286
ouvertures aux échecs 294

P

paradoxes 77
d'Epiménide 78
de Berry 78
de Russell 77
parallélisme 284, 377
PARRY 32
partial look ahead 253
partition des modèles 99
PCP 140
PECOS 379
perception 28
perceptron multi-couches 470
pièce chinoise 481
planarité d'un graphe 155
planification de tâches 165, 168
plus court chemin 155
plus général unificateur ..... *voir* logique des prédicats
poker 207
polymorphisme 198
portée 56, 177, 348
Post 135
postulat mécaniste 26, 27
prédicats ..... *voir* logique des prédicats
prémisses 48
principe de compositionalité 42
principe de déduction 93
principe de résolution 28, 102, 112
principe du tiers-exclu 79
principe minimax 268
principe négamax 268
probabilité d'arrêt 149
problèmes 152
Δ-voyageur de commerce 168
ϵ-approximation 166
2-SAT 163
3-SAT 163
à information totale 223
calculables 139–147
circuit hamiltonien 164
classe EXPTIME 172
classe NPSPACE 171
classe NP 156
classe PSPACE 171
classe P 154
classe co-NP 156
classe co-P 155
complémentaires 152
complexité 151–172, 222
conjecture d'Erdös 264
cycle eulérien 155
cycle hamiltonien 164
de décision 84, 152
de Post 36, 140
décidables 84, 139–147
décisionnel de Markov 448
décomposables 222
dixième de Hilbert 146
du Zèbre 265
faciles ..... *voir* problèmes, classe P
ignorables 223
indécidables 84, 139–148
intraitables 155
linéaires 154
missionnaires et cannibales 206, 210
NP-complets 94, 157
exemples 163
optimisation 224
ordonnancement 165
polynomiaux ..... *voir* problèmes, classe P
prévisibles 223
pseudo-polynomiaux 169
reconnaissance d'un mot 139
récupérables 223
réductibles 157
sac à dos 163
SAT 101, 158
satisfaction de contraintes ..... *voir* CSP
satisfiabilité 158, 224
schéma d'approximation 167
complet 167
polynomial 167
stable d'un graphe 164
transversal d'un graphe 164
voyageur de commerce 36, 164, 207, 211
processus décisionnel de Markov 445–453
équation de Bellman 450
équations d'optimalité 450
algorithmes 450
itération de la stratégie 451
itération de la valeur 451
programmation linéaire 451
real-time dynamic programming 452
RTDP 454
critères de performance 447
critère γ-pondéré 449
critère fini 449
fonction de valeur 448
partiellement observable 452, 472
probabilités de transition 447
stratégies 452
aléatoires 453

roll-out 451
stationnaires déterministes Markoviennes 448
programmation avec contraintes ..... *voir* programmation logique, avec contraintes
programmation des jeux ..... *voir* jeux
programmation dynamique .. *voir* processus décisionnel de Markov,algorithmes
programmation fonctionnelle ..... 175, *voir* ML
programmation génétique 439–441
programmation linéaire 228, 451
programmation logique ..... *voir* PROLOG
avec contraintes 378–385
contraintes 380
programme 380
propriétés 381
résolution 381
schéma CLP($\mathcal{X}$) 379
structure 380
PROLOG 28, 66, 104, 337, 363–385
backtrack ..... *voir* retour arrière
base de faits 364
base de règles 364
clauses de Horn 364
`cut` 375
évaluation gelée 376
faits 366
`freeze` 376
historique 363
liste 365
moteur d'inférence 365, 367
négation 377
parallélisme 377
points de choix 368
prédicats 365
pré-définis 376
question 365
règles 366
résolution 370
retour arrière 368
structure copying 373
structure sharing 373
unification 372
occur-check 373
test d'occurrence 373
variable 365
PROLOG III 382
contraintes 382
relations 382
propagation de contraintes ..... *voir* CSP, filtrage
propriété du diamant 181
PROSPECTOR 30, 37
psychologie 290–294
cognitive 485, 489

Q

question 365
Quine 94

R

R1/XCON 30, 37, 338, 339
Récupérateur de mémoire 348
radial basis functions 469
raisonnement déductif 208
real full look ahead 253
recherche en largeur 215
recherche en profondeur 214
recherche heuristique ..... *voir* fonction heuristique
recherche meilleur en premier 218
recherche secondaire 301
reclassement des coups 279
récompenses retardées *voir* apprentissage par renforcement
reconnaissance 298
recuit simulé 406
récursif 133
récursivement énumérable 133
réduction 95
régime de contrôle 334
règle de Hebb 397
règles d'inférence 28, 48
conclusion 48
prémisses 48
règles de production 136
règles du delta 401
règles du gradient 403
regroupement en hyper-arbre 259
relation de satisfiabilité 44
renommages de variable 178
représentation des connaissances 322
reproduction 416
réseaux de neurones 395–413
à deux couches 400
à sens unique 400
apprentissage non supervisé 410
apprentissage supervisé 401
axone 395
Boltzmann 406
dendrite 395
feedforward 400
fonctions d'activation 402
Kohonen 410
logiques 406
multi-couches 402
règles de Widrow-Hoff 400
règles du delta 401
règles du gradient 403
rétro-propagation 403
taux d'apprentissage 405
réseaux sémantiques 329
résolution de conflit 334
résolution de problèmes 205
résolvante 102
retour à l'équilibre 301
rétro-propagation 403
Reversi ..... *voir* jeux, Othello
RITA 338
robotique 474
RSA 171
Russell 77

S

sac à dos 163
Samuel 267, 474
SAT ..... 101, *voir* problèmes, SAT
satisfaction 228
satisfiabilité 44, 224
SBL ..... *voir* apprentissage, symbolique
schéma CLP($\mathcal{X}$) 379
schéma d'approximation 167
  complet 167
  polynomial 167
schéma de type 199
science cognitive ..... *voir* IA, cognitive
Scout ..... *voir* algorithmes
scripts 332
Searle 481
sélection 334
séquence de calcul 67
série aléatoire 148
Shannon 267
Shaw 267
SHRUDLU 338
Simon 267, 480
skolémisation 106
SMALLTALK 332
solution d'un CSP 232
solution déductive 208
solution inductive 208
SSS* ..... *voir* algorithmes
stable d'un graphe 164
stratégie 273, 300
  partielle 273
stratégie de contrôle 205
SU/X 338
substitution 46, 110, 178
  dans un schéma de type 200
  de renommage 111
  de types 196
  pure 110
  uniforme 136
support d'une valeur 238
système de Post 135
  axiomes 136
  calcul propositionnel 137
  forme normale 137
  langage généré 136
  purs 136
  réguliers 136
  règles de production 136
  substitution uniforme 136
  théorèmes 136
système de production 205, 208
  commutatif 209
  monotonique 209
  partiellement commutatif 209
système de réécriture 132
  dérivation 133
  règles de production 133
  relation de production 133
système de typage 196
système expert 27, 28, 37, 321–343
  à base de modèle 322
  à base de règles 333
  base de faits 333
  base de règles 333
  critiques 339
  moteur d'inférence ..... *voir* moteur d'inférence
système S4 ..... *voir* logique modale
système S5 ..... *voir* logique modale
système T ..... *voir* logique modale
système multi-agents 475

T

tables de hachage 280
tables de transposition 279
tables de vérité 44, 61
tactique 300
tautologie 45
taux d'apprentissage 405
taux de satisfiabilité 231
TDGammon ..... *voir* jeux,backgammon
termes du λ-calcul 176
  clos 177
  stratifiés 198
  typés 193
termes du premier ordre 56
  complètement instanciés 108
  de base 108
test d'occurrence 112, 373
test de Turing 31
tester et générer 236
théorème 48, 136
  de Church-Rosser 181
  de Cook 158
  de déduction 50, 59
  de Gödel (conséquences) 90
  de Gödel (consistance) 89
  de Gödel (incomplétude) 88
  de Herbrand 108
  de standardisation 182
théorie de l'information 147
théorie formelle 80
thèse de Church-Turing 66, 84, 187
tiers exclus 79
TMS ..... *voir* maintien de cohérence
traduction automatique 26
transformation polynomiale 156
transversal d'un graphe 164
tree clustering 259
Turing 31, 65, 267
typage inféré 195
typage polymorphique 199
types 192
  atomiques 192
  avec variables 195
  principaux 198
  utilisateur 355

U

unifiabilité 111
unificateur 111
  le plus général ..... *voir* logique des prédicats
unification 112
univers de Herbrand 108

V

valeur de vérité 43
valuation 44
variable d'état ..... *voir* espace d'états
variables 176, 365
de type 196
génériques 200
liées 56, 177
libres 56, 177
portée 56, 177
propositionnelles 42
scope 56, 177
voyageur de commerce 164, 167, 207, 211

W

Winston 481

Messages
111, rue Nicolas-Vauquelin – 31100 Toulouse
Tél. : 05 61 41 24 14 – Fax : 05 61 19 00 43
mél : imprimerie@messages.fr

Le but de cet ouvrage, à vocation pédagogique, est de dégager les contours souvent flous de ce que l'on appelle Intelligence Artificielle et d'aider à mieux comprendre quelle est sa place dans l'informatique moderne. Pour bien saisir l'ensemble des enjeux, il est bon de poser le problème en termes précis.

C'est pourquoi les bases théoriques de l'Intelligence Artificielle (logique et résolution) et les fondements de l'Informatique théorique (calculabilité, complexité, lambda-calcul) sont développés dans la première et la deuxième partie de ce livre. Puis les principales techniques de l'Intelligence Artificielle sont exposées de façon détaillée dans la troisième partie : programmation des jeux (du classique alphabeta aux jeux Conway en passant par SSS*), problèmes de satisfaction de contraintes, algorithmes de parcours d'arbres ou de graphes (A*, ...).

Les langages modernes de l'Intelligence Artificielle, qu'ils fassent partie de la programmation logique (Prolog) ou de la programmation fonctionnelle (Caml) sont présentés dans la quatrième partie. La cinquième partie est consacrée aux méthodes d'apprentissage symbolique, neuromimétiques et par renforcement, ainsi qu'aux algorithmes génétiques. La conclusion tente de replacer l'Intelligence Artificielle dans son contexte social et philosophique, et de comprendre comment elle a pu susciter tant de discours différents et tant de polémiques.

Cet ouvrage offre un panorama des connaissances théoriques et techniques indispensables pour bien comprendre l'informatique d'aujourd'hui et évoluer vers ce que sera l'informatique de demain.

Chez le même éditeur :

- *Techniques avancées du traitement de l'information 2e éd.*
J.-L. AMAT, G. YAHIAOUI

- *Simulation & algorithmes stochastiques - Une introduction avec applications*
N. BARTOLI, P. DEL MORAL

- *Programmation fonctionnelle appliquée à l'analyse matricielle*
A. COUTURIER, G. JEAN-BAPTISTE

- *Probabilités et statistiques appliquées*
B. LACAZE, C. MAILHES, M.-M. MAUBOURGUET, J.-Y. TOURNERET

www.cepadues.com

Réf. : 578
I.S.B.N. : 2.85428.578.6
9 782854 285789